ADAPTIVE SYSTEMS FOR THE DYNAMIC RUN-TIME OPTIMIZATION OF PROGRAMS

Gilbert Joseph Hansen

Carnegie-Mellon University

Prepared for:

Defense Advanced Research Projects Agency
Air Force Office of Scientific Research

March 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR-74-1436	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 784 880
4. TITLE (and Subtitle) ADAPTIVE SYSTEMS FOR THE DYNAMIC RUN-TIME OPTIMIZATION OF PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gilbert Joseph Hansen		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE March, 1974
		13. NUMBER OF PAGES 179
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research /NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis investigates adaptive compiler systems that perform, during program execution, code optimizations based on the dynamic behavior of the program as opposed to current approaches that employ a fixed code generation strategy, i.e., one in which a predetermined set of code optimizations are applied at compile-time to an entire program. The main problems associated with such adaptive systems are studied in general: which optimizations to apply to what parts of the program and when. Two different optimization strategies result: an ideal scheme which is not practical to implement, and

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (abstract cont.)

a more basic shceme that is.

The design of a practical system is discussed for the FORTRAN IV language. The system was implemented and tested with programs having different behavioral characteristics. In order to have a basis for comparing the results, variants of the system were constructed which approximate the behavior of WATIF, FORTRAN IV G, and FORTRAN IV H compilers. The test programs were run under these systems. The results show that adaptive FORTRAN performs as well or better than any of the variant systems at each specific test point, and significantly better than any one of them across the entire range of test points.

ADAPTIVE SYSTEMS FOR THE DYNAMIC RUN-TIME OPTIMIZATION OF PROGRAMS

Gilbert Joseph Hansen

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
March, 1974

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the office of the Secretary of defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ib

Abstract

This thesis investigates adaptive compiler systems that perform, during program execution, code optimizations based on the dynamic behavior of the program as opposed to current approaches that employ a fixed code generation strategy, i.e., one in which a predetermined set of code optimizations are applied at compile-time to an entire program. The main problems associated with such adaptive systems are studied in general: which optimizations to apply to what parts of the program and when. Two different optimization strategies result: an ideal scheme which is not practical to implement, and a more basic scheme that is.

The design of a practical system is discussed for the FORTRAN IV language. The system was implemented and tested with programs having different behaviorial characteristics. In order to have a basis for comparing the results, variants of the system were constructed which approximate the behavior of WATFIV, FORTRAN IV G, and FORTRAN IV H compilers. The test programs were run under these systems. The results show that adaptive FORTRAN performs as well or better than any of the variant systems at each specific test point, and significantly better than any one of them across the entire range of test points.

Acknowledgements

My sincere appreciation and thanks go foremost to Professor William A. Wulf, my thesis advisor, who originally suggested this topic, and who provided constant inspiration and guidance throughout its development.

I am also indebted to Professors Mary Shaw, Jack McCredie, and John Grason, members of my thesis committee, who helped shape the final form of the thesis.

Finally, special thanks are due to the Department of Computer Science for providing the excelient facilities and stimulating atmosphere in which to carry out the research.

Table of Contents

Chapter I

Introduction

A serious disadvantage of current compilers is that they do not take into account a program's behavior in the generation of object code. In particular, the code generation phases of these compilers employ a fixed compile-time strategy, i.e., the degree of code optimization is predetermined and the optimizations are applied uniformly to each section of a program, independent of how often the section is executed. As a consequence, special purpose compilers have been designed to handle a specific class of programs or to meet specific needs, and the decision of which compiler to use is placed upon the user. For example, for the FORTRAN language there exist on the same machine three special purpose compilers having different trade-offs between compile time and code efficiency, viz., WATFIV, FORTRAN-IV G and FORTRAN-IV H. WATFIV is designed to handle jobs for which compile time is a major factor. FORTRAN-IV G produces fairly efficient code by applying some local optimizations. FORTRAN-IV H is designed for production programs. It produces highly efficient code, but there is a substantial increase in compilation time.

This thesis investigates a dynamic run-time code optimization strategy based on the dynamic behavior of the program. Motivation for such a system stems from the empirical evidence produced by the research of

Knuth [Knu70], Darden and Heller [Dar70], Ingalls [Ing71] and Jasik [Jas71], viz., that a small part of a program (<5%) accounts for a large part of its execution time (>50%). Their schemes can be classified as "iterative optimization" which involve a feedback loop between the system and the user. The user's program is monitored via software or hardware, and the system produces an execution profile of where the program is spending its time. Using this profile, the user optimizes his program and runs it again, obtaining another profile; and so forth. Major drawbacks to such an approach are the limitations placed on the amount of optimization the user can perform, and the inclusion of the user in the feedback loop. We advocate removing the user from this feedback loop and automating the process.

Our major goal is to demonstrate not only that it is possible to construct such an automated system, but that it is worthwhile, i.e., that it can perform, for almost all programs, as well or better than any special purpose compiler employing a fixed code generation strategy.

1.1 Current Optimization Techniques

The development of code optimization strategies has been under investigation since 1965. This initial research culminated in a set of machine independent optimizations that are applicable to most high level languages [cf. All69]. The development of more efficient algorithms for these "classical" optimizations has been the object of study by other investigators, notably

Lowry and Medlock [Low69] and Cock and Schwartz [Coc70]. The effectiveness of these optimizations was clearly demonstrated in the FORTRAN-IV H compiler of Lowry and Medlock, who stated that even though there was a 40% increase in compilation time, the object code was 25% smaller and executed three times faster than that produced by the FORTRAN-IV G compiler.

The goal of this research is to demonstrate the effectiveness of applying code optimization at run-time instead of at compile-time. It suffices to select optimizations from among the "classical" optimizations, for they are just as applicable at run-time. There were a number of selection criteria that are worth mentioning. Foremost, we wanted to include enough machine independent and dependent optimizations to produce results of broad significance. Also, the optimizations must have been proven by others to be effective, i.e., they produce a significant decrease in execution time for the effort expended.

The set of machine independent optimizations selected were:

1) Constant Folding: performing operations whose operands are known. This technique is particularly beneficial for code generated to calculate the address of an array element.

2) Common Subexpression Elimination (CSE): eliminating redundant expression computations.

3) Code Motion (CM): moving operations invariant within a loop outside the loop.

The set of machine dependent optimizations selected (which are applicable to most machines) are:

1) replacing a multiplication or division by a power of 2 with a shift.

2) setting memory to 0 or -1 by special instructions.

3) delaying negation operators to exploit load and store negative instructions.

4) deleting multiplications by 1 or additions of 0.

5) performing operations directly to memory, e.g., incrementation or decrementation by a small constant.

6) use of index registers for DO loops and for accessing array elements.

7) effective use of registers by an appropriate register allocation policy.

The algorithms for the selected optimizations have certain characteristics that influence the design and structure of any system which employs run-time optimization. First, the algorithms do not operate on the program source text, but on some intermediate form. The compiler must generate this intermediate form (regardless of when the optimizations are applied). Since these optimization algorithms are to be invoked at run-time, the intermediate form was chosen so that it could be directly executed (interpreted).

Second, the algorithms do not operate at the basic instruction level, but on aggregates of instructions or groups of aggregates (loops). The compiler will have to decompose the program into these basic aggregates.

Third, certain algorithms rely on control flow analysis. The compiler (or loader) will have to generate a form for encoding the flow relationships. The form we will use is a directed graph.

Finally, the optimizations can be applied individually, and usually must be applied in a given order. These two characteristics are important in that they allow for gradual optimization of the program, a concept fundamental to our approach which is predicated on and supported by recent empirical results on program behavior.

1.2 Empirical Results on Program Behavior

Recent investigations by Knuth [Knu70], Ingalls [Ing71] and Darden and Heller [Dar70] found that a small portion of the code in typical programs accounted for most of the execution time. Knuth studied a varied collection of FORTRAN programs covering a wide variety of applications, and found that less than 4% of a program accounts for more than 50% of its execution time. He suggested that the system produce a program's profile, i.e., a histogram showing the frequency counts of the executable statements, which can reveal where the program is spending its time. This information would then be used by the user or compiler in deciding what part of the program to optimize.

Ingalls participated in Knuth's investigation and his paper pursues the notion of a system producing the execution profile of a program. He

concludes that current optimizations have taken us about as far as is worthwhile, and that if further gains are to be made, optimizations such as in-line I/O editing or expansion of subroutines in-line should be developed, or the sytem should produce feedback information (i.e., an execution profile) to the user that tells him where his program is spending most of its time. He found that for all the programs studied, 3% of the statements made up 50% of the program's execution time.

Darden and Heller studied the performance of two compilers and an assembler, and found that for the systems tested, at most 3% of the code accounted for more than 60% of the execution time. These percentages are taken from their graphs given in Figure 1.1. They advocated producing a histogram of processor time by blocks of memory locations. Using this profile, the user would optimize the critical sections of the code and run the system again. This iterative optimization procedure would be repeated until there was little improvement in overall performance. They tried this technique on an ALGOL compiler and found that after four iterations they had improved the compiler's speed by a factor of 10 while only rewriting 5% of the code.

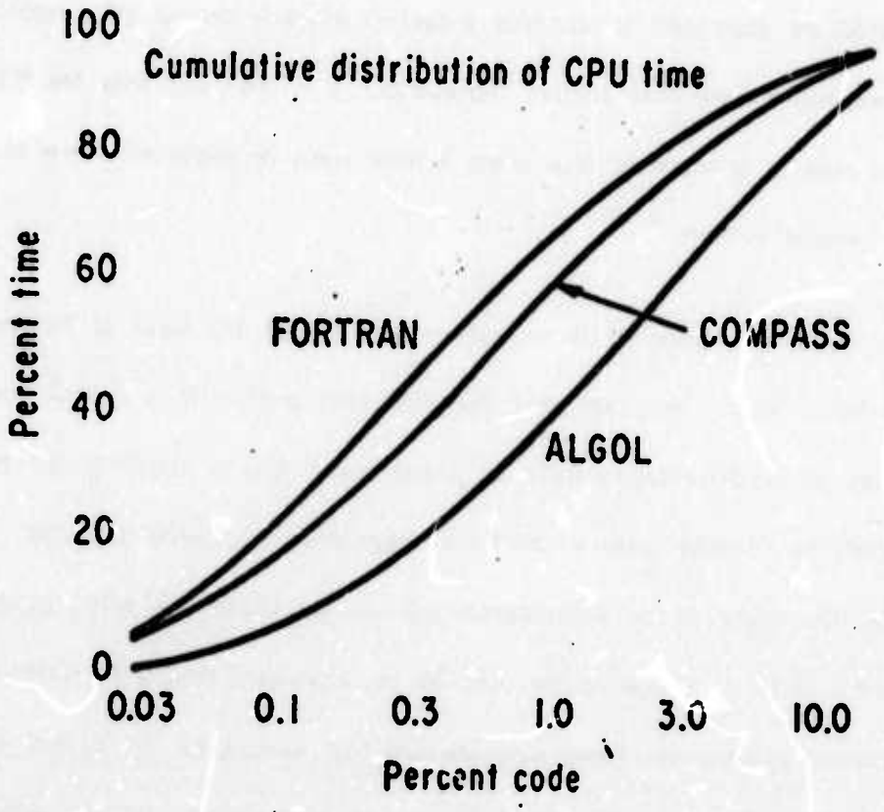


Figure 1.1 Cumulative distribution of CPU time. For a typical FORTRAN compiler, over 60 percent of the central processor's time is spent in executing only 1 percent of the code. Clearly, that 1 percent of the code is the area to optimize. In fact, 10 percent of the code accounts for 90 percent of the execution time of all the systems tested by the authors. (Reprinted from COMPUTER DECISIONS, October, 1970, page 29-33, copyright 1970, Hayden Publishing Co.)

An inference from these empirical results is that the amount of effort that should be expended to optimize a section of code should be proportional to the execution time that section represents. It is also felt that the 5%-50% empirical rule is universally true since a wide class of problems were studied by the various authors.

A major drawback to these approaches is that the user is included in the feedback loop. We feel that the execution profile is a useful concept which has other advantages (such as a debugging aid, or pointing out to the user that he should use a different algorithm or restructure his data). However, its utility as an optimization tool has its limitations with respect to the user. First, it requires the user to be knowledgeable with optimization techniques. Second, for those optimizations that cannot be performed at the source level, the user must resort to writing in machine language which he must learn, thus defeating the purpose of using a high level language. Finally, the user may introduce more bugs into the program.

The user could overcome these limitations, but we assert (and this thesis will show) that the process of using the execution profile to optimize the appropriate sections of the program can be done automatically at the source language level without user intervention.

1.3 An Adaptive Compiler

There are a number of automated approaches that we could take. The profile could be fed back to the compiler the next time the program is compiled. Using the profile, the compiler could optimize the appropriate sections of the program. However, which code sections to optimize may vary from run to run if, for example, the program's behavior is sensitive to its input data. Therefore, a feedback system does not seem to provide the best solution.

A more desirable approach would be to perform code optimizations while the program is running. That is, the system would dynamically adapt the compiled code in response to the program's dynamic behavior. Such a system we term an **adaptive† system**.

This thesis will show that an adaptive compiler system is a feasible and worthwhile alternative to current compiler construction approaches. We will first turn our attention to solving the problems of determining which section of code to optimize, when to optimize it, and how much optimization to apply to it. Our goal is to find a solution that minimizes the overhead incurred in answering these questions, for we want the system to perform well across the entire execution time spectrum. Then, in order to prove the technique is

---

† The term adaptive is not meant to imply that the compiler self-adapts to its environment, i.e., keeping statistics on the constructs used most frequently and thereby producing more efficient code for them.

feasible, we will discuss the design and implementation of an adaptive system for the FORTRAN-IV language. To show the adaptive FORTRAN system is worthwhile, its performance will be measured on a variety of test programs having different characteristics. In order to evaluate the performance measurements in a meaningful and unbiased manner, the adaptive compiler will be transformed into systems that generate code comparable to that produced by WATFIV, FORTRAN-IV G and FORTRAN-IV H. Then the test programs will be run under these systems and the performance measurements compared with those of the adaptive system.

Chapter II

Adaptive Compiler Systems

In this chaper we will look at the problems associated with constructing an adaptive compiler and present solutions that can be realistically implemented. The basic issues that we will address are:

1) what information must be collected during the translation and loader phases to facilitate run-time optimization,

2) the characteristics of the code the translator must produce for the optimizers,

3) methods for grouping the code into blocks to facilitate its processing by the optimizers,

4) attributes of code blocks that can be metered to determine which blocks to optimize,

5) methods for determining which optimizations to apply to the code blocks and when,

and 6) control of the running program so optimization can be intermixed with execution.

Three of the issues, viz., determining which code blocks to optimize, when, and how much, form the basis for any dynamic optimization strategy. We will discuss two strategies. The first, iterative dynamic optimization (see Section 2.4), is based on a mathematical model, which represents an exact formulation for solving the problem of what to optimize and how much, but not when. The scheme is impractical to use (see Section 2.4.2), but it is presented because the solution of such a formulation, regardless of how

inefficient, would give us a standard to compare against other schema. It is possible to obtain such an absolute measure of performance (see Chapter 5), but even for one program it would require a tremendous amount of work. Since dynamic optimization has never been studied before, it was felt that the primary goal of the thesis was to see if the approach was vaild instead of spending time to obtain the best performance curve for a few programs. Therefore, we formulated a more direct approach, the incremental dynamic optimization scheme (see Section 2.5), which incurs very little overhead. It is a heuristic approach based on the notion that one optimization at a time should be applied to a code block, and the assumption that the execution time per program run for a code block is proportional to the frequency with which it is executed. Such an assumption allows a frequency count to be used as a metric for controlling the adaption process. This count is a function of the code block's attributes, such as size, level of nesting, etc..

To control the execution and invoke an optimization strategy requires a supervisor. We will describe the operational characteristics for such a supervisor in general, and specifically for a system employing the incremental dynamic optimization strategy.

In the following chapters, we will describe the design, implementation and performance of an actual system that employs incremental dynamic optimization and incorporates the ideas expounded in this chapter. Since the iterative dynamic optimization scheme in Section 2.4 is not pertinent to this

description, it may be bypassed on a first reading without loss of continuity.

2.1 Overall Design Considerations

The primary goal in the design of an adaptive compiler system is to minimize the total cost of running a program. This goal has direct implications with respect to the design of the translator and the dynamic optimization strategy.

The design of the translator can proceed along the lines currently employed in the construction of any translator, but it must be as efficient as possible. This means that: 1) it should expend a minimum of effort in translating the source code, in particular, not performing any optimizations that can be done more effectively and efficiently at run-time; 2) it should employ the best translating algorithms available; 3) it should itself be optimized; and 4) it should be one pass† and compile directly to core.

In terms of the dynamic optimization strategy, minimizing total cost requires the optimization algorithms to be efficient, and the overhead incurred by deciding when to perform what optimization on which sections of code to

---

† A second pass over the object code is needed to complete the translation process, e.g., allocate data storage, patch addresses, patch forward references, relocate the object code, etc.. This second pass of the compilation process is handled by a program which, due to its similarity to others of the same name, we shall call a loader.

be small compared to the expected payoff.

There are four basic design decisions that must be made; they are a consequence of both the fact that optimization is to be performed at run-time and the nature of the optimizers. First, an internal representation of the source code that can be efficiently manipulated by the optimizers must be selected. This internal form cannot be machine language because at this level too much information that will be needed by the optimizers has been lost, and it should not be the source code because the source code does not explicitly indicate the structure of the program and takes too much time to scan. Possible internal forms include: Polish notation, quadruples, triples, indirect triples, or trees (cf. [Gri71]). The translator can produce the internal form as object code, for it is amenable to being executed (interpreted). For the translator also to produce machine language would be a waste of effort because empirical evidence shows that some sections of code will not be executed often enough to warrent it.

Second, the internal form (which we will assume is an n-tuple that denotes an "instruction") must be grouped according to the program's structure into aggregates so that the optimizations which require global flow information can be applied. It is a characteristic of the classical optimizations that we will employ (see Section 1.1) that they operate on two kinds of aggregates: a group of sequential instructions terminated by an unconditional branch (a basic block) and a group of basic blocks which form a loop-like

structure (a segment). Initially, as the internal form is being generated, it is partitioned by the translator into basic blocks. As the program executes, optimizations are performed on those basic blocks executed most frequently. In order for additional optimizations to be performed, the segment containing an optimized basic block must be formed. The process of combining basic blocks or segments into a (larger) segment is called fusion, and constitutes a new optimization.

Third, a dynamic optimization strategy must be proposed, i.e., a scheme for determining which basic blocks or segments to optimize, which optimizations to apply and when to perform the optimizations. Even though there is more information available at run-time than at compile-time to aid in making these decisions, it is not complete (we cannot predict a program's future behavior with absolute accuracy). A reasonable approach is to assume that future behavior of a program will be similar to past behavior, for it is better to base the decision-making on this information than none at all. Such an assumption is not uncommon; it is often made in other areas of computer science (e.g., paging algorithms and schedulers). As is the case with the other areas, we are susceptible to anomalies. For example, it is possible to waste optimization effort if the program terminated soon after the effort was expended, or the program is phased and after the optimization of a phase it is only executed a few times more and then never executed again. By selecting a strategy that causes optimization of basic blocks or segments to be gradual, the amount of effort wasted can be kept tolerably small.

Finally, means for controlling the execution of a program so it can be adapted must be determined. Basic blocks and segments are the discrete units of execution. A logical point at which to interrupt a program's execution for adaption is when control passes between two basic blocks or segments, since program status is well defined at such points, and the amount of state information required to record this status is small. When execution is interrupted, data which aids in the decisions made by the dynamic optimization strategy is collected, and it is decided whether to invoke the dynamic optimization strategy and perform optimizations. If control of execution is distributed amongst the individual basic blocks and segments, then appropriate instructions must be inserted in the code to perform the functions just described. Another approach is to centralize these functions and control of execution in a supervisor which causes the basic blocks or segments to be executed one at a time. This is the approach we will follow for the system to be implemented. The supervisor, known as the **segment driver**, is advantageous for another reason. During the execution of the program, some parts of it will be in interpretive code, while other parts will be in machine language. The supervisor can conveniently decide whether to execute a basic block or segment directly or call an interpreter.

The structure of an adaptive compiler system is now apparent. Source code is translated by a fast and efficient translator into an internal form that is grouped into basic blocks. Execution of the program is controlled by a segment driver and optimization by a dynamic optimization strategy. Various

optimizations are applied to basic blocks and/or segments as execution proceeds and the performance of the program warrents. A new optimization, fusion, is necessary for grouping basic blocks into segments. In the following sections, we will define more precisely the concepts of basic block, segment, fusion, and segment driver, and propose two dynamic optimization strategies.

2.2 Basic blocks and Segments†

When performing code optimizations, it is advantageous to partition the program according to its flow of control into basic blocks. A **basic block** is a linear sequence of instructions with the first instruction being the single entry point. The block is terminated by one or more branch instructions, the last of which is unconditional while the others (if any) are conditional. All code between the first instruction and the branches is executed in sequence.

A program's flow of control may be represented by a directed **control flow graph** in which a node represents a basic block and an edge represents a flow path. Those basic blocks that branch to a given block are its

---

† In this section, some of the definitions follow the terminology introduced by Allen [All69, All70] (viz., basic block and directed control flow graph) and Lowery and Medlock [Low69] (viz., predominance), while others pertaining to directed graphs can be found in any introductory textbook on graph theory (cf. [Har69]) (viz., immediate successor or predecessor, subgraph, path and strongly connected region).

immediate predecessors. Likewise, those blocks branched to by a given basic block are its immediate successors. A basic block may have more than one immediate predecessor or successor, including itself. Program entry blocks have no predecessors, and program terminating blocks have no successors. A basic block B_1 predominates a block B_2 if every path along a sequence of successors from a program entry block to B_2 always passes through B_1.

The basic block is the smallest program unit commonly considered for optimization. However, there is a limit to the amount of optimization that can be performed on a basic block, and in order to perform additional optimization it is necessary to consider more global context. Since it is desirable to optimize those basic blocks executed repetitively, some loop-like structure must be imposed on the flow graph. Two loop-like constructs have been described in the literature: the strongly connected region [All69] and the interval [Coc70, All70]. A strongly connected region is a subgraph of the flow graph in which there is a path leading from any block in the region to every other block. The region may have several entry points. An interval is the maximal single entry subgraph of the flow graph in which all closed paths contain the entry block.

We introduce another similar, but not equivalent, concept called a segment. With respect to a given strongly connected region (loop) in the directed control flow graph, a segment is the minimal directed subgraph with the following properties:

1) The segment contains all the basic blocks in the loop.

2) There is a single entry block. This entry block may have one or more immediate predecessors of which at least one is not contained in the segment. Those immediate predecessors of the the entry block that are predominated by it are contained in the segment.

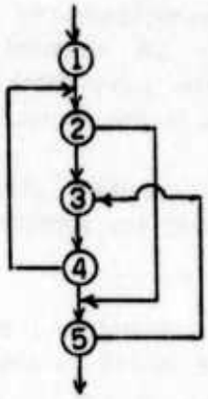
3) Except for the entry block, all immediate predecessors of each basic block in the segment are contained in the segment.

4) The segment, A_i, and another segment, A_j, are either disjoint, i.e., they have no basic blocks in common and therefore are parallel structures, or one is embedded inside the other. If $A_i \cap A_j = A_j$, then A_j is embedded inside A_i, and A_i is said to cover segment A_j.

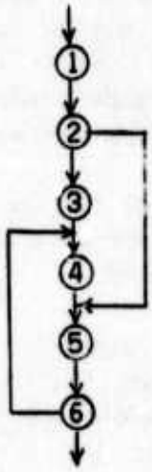
Thus, if a loop has a single entry block, it is identical to a segment whose segment entry block is the same as the loop entry block. If the loop has multiple entry points, the segment is the loop extended to include the minimum number of basic blocks satisfying properties 2, 3 and 4. Property 4 defines a properly nested set of segments, and allows the optimizations to be ordered in the manner suggested by Allen [All69].

Examples: (a) Segments: 2'={2,3} (b) Segments: 2'={2}, 2''={2,3}

(c) Segments: 2'={2,3,4,5}



(d) Segments: 2'={2,3,4,5,6}



Unlike a strongly connected region, there is not necessarily a path leading from any block in a segment to any other block because of the requirement that a segment have a single entry point. A segment always contains a strongly connected region, but the single entry point for the segment may not necessarily be contained in the region. Consider example (d) above in which segment 2' contains the strongly connected region {4,5,6} which has two entry points (blocks 4 and 5), and the entry point of the segment (block 2) is not contained in the region. Like a segment, an interval contains a single entry point, but it does not necessarily contain a closed path, and the intervals of a graph are disjoint. In example (b) above, the intervals are {1} and {2,3}.

The concept of a segment was chosen over that of a strongly connected region or interval because of the simplicity of the algorithm for constructing them. The algorithm is an iterative process. A block which is

executed repetitively can be used to start the segment. Given such a block, and considering a segment already formed as a basic unit, it is possible to construct the segment containing the block knowing just the immediate predecessors of each basic block in the flow graph. A basic block's list of immediate predecessors can be constructed from the branches that terminate each basic block. This can be done either by the compiler or by the loader.

2.3 Fusion

When a segment is formed via fusion, what optimizations are applicable to it depend on how embedded segments are treated, i.e., whether or not the new segment is considered to be a homogeneous structure with respect to future optimizations. Define the **optimization state** of a basic block/segment to be the result of the application of an optimization. As optimizations are performed on a basic block/segment, they will advance through different optimization states. One basic block/segment is said to have a higher optimization state than another if more optimizations have been applied to it. If we employ **homogeneous fusion**, then the optimization state of a segment is uniform, i.e., is the maximum optimization state of its constituents, and the result is a **homogeneous segment**. If the segment contains no embedded segments, then its optimization state is the maximum optimization state of its basic blocks; otherwise it is the maximum optimization state of the segments it covers. In order to advance the segment to its optimization state it may be necessary to perform one or more optimizations on its embedded basic

blocks/segments. For future optimizations, the covering segment could be considered a discrete unit. But this has a distinct disadvantage if at the time of fusion the embedded basic blocks/segments have not attained the highest optimization state, for the effect of additional optimizations on these embedded units will never be realized. Alternatively, if the identity of the embedded basic blocks/segments is retained, further optimization could be applied to them before being applied to the covering segment. The only constraint is that all units attain the same optimization state.

If on the other hand we employ **non-homogeneous fusion**, a segment and its embedded basic blocks/segments exist at different optimization states, and the result is an **non-homogeneous segment**. This approach is more restrictive from an optimization point of view in the sense that the optimizations that are applicable to a segment and the segments it covers depend on the current optimization state of each. There are a number of ways optimizations can take place. One method is to let the covering segment control when the embedded segments get optimized further. For example, when a segment is to be optimized, its embedded segments could first be advanced to their next optimization state, starting with the innermost one and working outwards. Other approaches are to let each segment be optimized separately at its own rate, or to freeze the optimization state of each embedded segment at the time of fusion and let future optimizations be applied only to the covering segment.

The result of fusion is a machine language segment that is formed by combining the machine language for each of the segment's basic blocks. The machine language segment is to be considered a basic unit with respect to execution, i.e., transfer of control between basic blocks within the segment should not be processed by the supervisor. To accomplish this, branches which terminate a basic block must be treated differently depending on whether they are internal or external to the segment. An internal branch is a branch in which the basic block being branched to (destination) is in the same segment as the basic block containing the branch (source), and the destination is not the segment entry block; otherwise it is an external branch.

If a branch is internal to the segment, then it can be eliminated if it is to an immediate successor; otherwise it can be performed directly. If the branch is external to the segment, it must go through the supervisor. Once the segment becomes totally optimized, any branches to its entry block can be made directly.

When forming the physical segment, it is not clear whether to perform homogeneous or non-homogeneous fusion. We will defer this discussion until the next chapter where we will present empirical evidence as to the merits of each. There is, however, a general observation that we can make. When a segment is formed by fusion, the segments that will be included in it (if any) will have reached a high optimization state, if not the highest. This follows from the fact that embedded segments execute at a greater frequency

than their covering segment. Therefore, fusion that produces a homogeneous segment may tend to apply too much optimization too soon, while producing a non-homogeneous segment causes the optimization of the segment to be more gradual.

Fusion is thus seen to be an important optimization, for it determines the highest optimization state attainable for the segments involved, and thus has a strong influence on a program's performance. How to incorporate fusion into the overall optimization scheme is another of the basic problems in controlling dynamic optimization. We now present two dynamic optimization schemes based on different metrics that treat fusion differently.

2.4 Iterative Dynamic Optimization

The first dynamic optimization scheme is based on a cost metric, the total run-time cost associated with executing a program which is being adapted. This cost consists of the execution cost plus the optimization cost, and is a function of time and storage space. It includes optimization costs in order to guarantee that optimization will be gradual and performed when it pays to do so. Informally, we want to minimize the total run-time cost for a program. This means interrupting the program's execution periodically and, by knowing its past behavior, determining how it should have been optimized so that the total run-time cost would have been less than it actually was.

The scheme considers only the optimization of segments (which are determined prior to the start of execution), not of basic blocks. It is a non-homogeneous optimization scheme in which the rate of optimization for segments is free to vary, i.e., there is no restriction on the number of segments that can be optimized at one time, or the number of optimizations that can be applied to each segment. For the latter, we make the restriction that there be no backtracking, i.e., once a segment is optimized, it cannot be "deoptimized" back to what it was previously. The goal is to find the combination of optimizations that minimize the cost metric. This approach is a natural way to proceed, and has the advantage that fusion is not an issue (and therefore simplifies the formulation).

2.4.1 A Mathematical Model for Segment Optimization

Let

$$A=\{A_j\},\ j=1,2,\ldots,N\ .$$

be the set of segments for the directed control flow graph of a program P. Suppose there exists an ordered set of separate and distinct code optimizations

$$\{O_j\},\ j=1,2,\ldots,m$$

These optimizations are known as **singular optimizations**, and are ordered in the sense of applicability, i.e., O_1 must be applied before O_2, O_2 before O_3, etc.. The **composite optimization** $O_{ij}(i<j)$ is the transformation $O_j\ldots O_i$ which can be applied to a segment only if the transformation $O_{i-1}\ldots O_1$ has already been

applied.

The result of the application of one or more optimizations to a segment A is called a **representation**, R(A), of the segment. For a segment $A_i \in \mathcal{A}$, the only possible ordered representations that it may attain are:

$$R_1(A_i) = O_1 A_i$$
$$R_2(A_i) = O_2(O_1 A_i) = O_{12} A_i$$
$$\vdots$$
$$R_m(A_i) = O_m(\ldots O_2(O_1 A_i))\ldots) = O_{1m} A_i$$

R_i is said to have a higher optimization state than R_j if $i>j$.

The current representation of a segment is the result of the application of the composite optimization O_{1j}, for some j. If the current representation for segment A_i is $R_j(A_i)$, then for additional optimizations the segment is constrained to take on a new representation $R_k(A_i)$, where $j \leq k \leq m$. Not all new representations are possible. A **feasible representation** is one which does not violate the constraint that the optimizations are ordered. If one segment is not covered by another, then there is no restriction on what new representation it can attain. However, if one segment covers another, the covering segment cannot be optimized such that its otimization state is greater than that of the embedded segment. The new representation for the segment defines the optimizations that must be applied. That is, since the composite optimization O_{1j} has already been applied, it is only necessary to apply the composite optimization O_{jk} (if j=k, than this is the null optimization).

Consider a subset $A \subset \mathcal{A}$ which consists of $n \leq N$ segments executed since optimization was last performed. Define the n-tuple $R = \langle R(A_1),\ldots,R(A_n)\rangle$ with the property that each $R(A_i)$ is a feasible, ordered representation of equal or higher optimization state than the current representation of A_i. The goal now is to find the n-tuple R that minimizes the cost metric.

Let $C_i(R(A_i))$ be the cost associated with segment A_i being in representation $R(A_i)$, i.e.,

$$C_i(R(A_i)) = C1_i(R(A_i)) + C2_i(R(A_i))$$

where $C1_i$ is the cost of executing segment A_i in representation $R(A_i)$,

$C2_i$ is the cost of changing segment A_i's current representation to $R(A_i)$.

The exact forms of C1 and C2 depend on how computer resources are accounter for, but in general they are a function of execution time, T, and core space, S. As a concrete example, consider the case where a user is charged for how much processor time he uses and (just) the core he uses. The cost associated with that part of core which is fixed is a constant that can be ignored. This fixed storage includes the run-time support package, data storage (since we are only considering code optimizations), and the interpreter†. In order to simplify the model, we ignore the time required to allocate and release core and to perform overlays. The form of C1 is then

$$C1_i(R(A_i)) = K_1 * T_i(R(A_i)) + K_2 * S_i(R(A_i)) * \sum_{k=1}^{n} T_k(R(A_k))$$

where K_1 is the cost for processor time,

K_2 is the cost of core storage used per unit of time,

$T_i(R(A_i))$ is the time expended executing segment A_i in representation $R(A_i)$,

$S_i(R(A_i))$ is the amount of core needed to store the representation $R(A_i)$ of segment A_i.

$C1_i$ is that fraction of the total cost for segment A_i, i.e., the sum of its processor cost plus its core storage cost (the summation term represents total execution time).

The form of C2 is similar to that for C1, but now we need to know the time to perform the transformation and the space occupied by each optimizer. These optimizer-dependent parameters are easily obtained once the optimizers are programmed. Suppose for each optimization, O_i, there exists a function $E_i(q)$ which gives the execution time to perform the optimization on a section of code consisting of q basic units, where a basic unit is related to the internal form and may be the number of nodes in the tree, the number of tuples, etc.. Let $S(O_i)$ be the amount of core needed to store the optimizer that performs optimization O_i. If segment A_i consists of q_i basic units, then

$$C2_i(R(A_i)) = C3_i(O_i)$$

---

† The interpreter is always assumed to be in core because it simplifies the formulation and it is highly likely that there will be at least some part of the program to be interpreted (as evidenced from the empirical result on program behavior).

where $C3_i(O_j)$ is the cost to perform optimization O_j. If O_j is the singular optimization O_j, then

$$C3_i(O_j) = K_1 * E_j(q_i) + K_2 * S(O_j) * E_j(q_i) .$$

If O_j is the composite optimization O_{1j}, then

$$C3_i(O_{1j}) = K_1 * \sum_{k=1}^{j} \delta_k * E_k(q_i) + K_2 * \sum_{k=1}^{j} \delta_k * S(O_k) * E_k(q_i)$$

where $\delta_k = 0$ if O_k has already been applied to A_i; otherwise 1. That is, if the current representation of segment A_i is $R_v(A_i)$, then the cost associated with the composite optimization O_{1v} is zero, and $C3_i(O_{1j})$ is just the cost of performing the composite optimization O_{vj}, $j>v$. These equations assume the optimizers reside in core only while they are needed, i.e., that they are overlayed.

The total cost associated with a program in which segment A_i is in representation $R(A_i)$ is

$$C = \sum_{i=1}^{n} C_i(R(A_i)) \qquad (1)$$

The objective is to find an n-tuple, R, of representations such that C is a minimum, i.e., solve

$$\min_{R} C \qquad (2)$$

subject to the constraints:

$$T_i(R(A_i)) \geq 0 \qquad (3)$$

$$\sum_{i=1}^{n} S_i(R(A_i)) \leq S \qquad (4)$$

where S is the total amount of available core storage. Constraint (4) requires that the new representations, R, for the segments all fit in core simultaneously.

An n-tuple of representations that solves (2) subject to the constraints (3) and (4) is known as the optimal policy with respect to the set A for executing P. The initial optimal policy for executing P is to perform no code optimizations and interpret the internal form produced by the compiler. Such a policy is in keeping with the philosophy of dynamic optimization (see Section 2.1). After P has executed for a while, a new optimal policy is determined according to (2). This policy is put into effect, and P allowed to continue execution. Later on, P's execution is again interrupted and a new optimal policy determined. This process is continued until P terminates.

2.4.2 Practicality of Using the Model

The iterative dynamic optimization strategy has three serious disadvantages which make it impractical to use. First, it only determines which segments to optimize and what their new representations should be, not when to determine a new optimal policy. Second, there is the problem of obtaining a numeric solution to (2). Any algorithm for solving (2) must be such that the total solution time expended during a run is a small fraction of the total execution time. To the best of our knowledge, the only way to solve (2) is by a combinatorial search, which tends to be time consuming. Third, there is the inability to generate the required data. In optimizing the

cost, it is necessary to determine the execution time, $T_i(R(A_i))$, and space requirements, $S_i(R(A_i))$, for the new representation $R(A_i)$ of segment A_i. Since the effect of an optimization on a segment cannot be ascertained without actually performing the optimization, T_i and S_i have to be predicted. This is undesirable because a policy so determined is only as good as the predictions. However, if the program's behavior is known a priori, it is possible to solve (2) and obtain an absolute measure of performance (see Chapter 5).

The model is therefore of more use in determining a standard against which other schema can be compared than being used in practive. We now present a more practcal approach in which the rate of optimization is more gradual than for the iterative dynamic optimization scheme.

2.5 Incremental Dynamic Optimization

The incremental dynamic optimization scheme is based on the assumption that the total execution time for a basic block/segment is proportional to the frequency with which it is executed. This assumption allows a frequency count to be used as a metric for deciding not only which basic block/segment to optimize, but when to apply optimization. Each time the basic block or segment is executed, this count is incremented†. When it exceeds a predetermined threshold, the basic block or segment is advanced to the next

---

† In practice, the count is decremented until it becomes negative.

representation by the application of the next optimization. Therefore, optimization is applied incrementally, i.e., one optimization at a time to one basic block/segment at a time. Fusion is automatically handled by this scheme since it is just one of the possible optimizations.

Define the **optimization count** for an optimization to represent the number of times a basic block or segment is to be executed in its current representation before applying the optimization (this is the threshold alluded to above). The optimization count associated with a basic block or segment must have the properties that it is proportional to the basic block/segment's execution time, and it determines the proper time at which to optimize the basic block/segment. Therefore, an optimization count will not be the same for each basic block/segment. Instead, it will be some function of the basic block/segment's characteristics, such as the length of the basic block/segment measured in some appropriate units, the basic block/segment's level of nesting in a loop structure, or the amount of effort required to apply the next optimization.

The optimization counts will be determined empirically. First, they will be estimated and treated as constants, then an empirical study made to determine what function of the basic block/segment's characteristics is most appropriate.

As an example of a possible function to study empirically, consider the following method for deriving optimization counts. Assume time is a

measurement of effort, and a basic block/segment consists of q basic units, where a basic unit depends on the internal form, e.g., for trees a basic unit is a node, for n-tuples, an individual n-tuple, etc.. Suppose for each optimization, O_i, there exists a function $E_i(q)$, which represents the time to perform the optimization on a basic block or segment consisting of q basic units, and t_{jk} is the time to execute basic block/segment A_j once in the ordered representation R_k. Then an estimate of the optimization count, n_{ji}, for basic block/segment A_j in the ordered representation R_i is

$$n_{ji} = E_i(q_j)/t_{ji-1} \quad , i=1,2,...,m$$

where m is the total number of distinct optimizations, and t_{j0} is the time to interpret the basic block/segment once. n_{ji} is the number of times the basic block/segment A_j can be executed in representation R_{i-1} before its total cummulative execution time is the same as the time it would take to perform the next optimization O_i. n_{ji} represents an upper bound because it would be wasteful to spend more time executing the basic block/segment than it would take to optimize it. Therefore the actual optimization count used should be some fraction of n_{ji}.

The quantity t_{jk} must be estimated. When the basic block/segment is first executed, let the supervisor clock its execution. This measurement is exact for a basic block because its execution is sequential. But for a segment it is an approximation since segments contain loops and internal branching. Therefore, we can assume the segment's timing to be exact only if we assume its future behavior will be the same as its past behavior.

Knowing t_{jk}, the supervisor can now calculate the basic block/segment's optimization count.

2.6 The Segment Driver

Optimization and execution of a program are under the control of the segment driver. Execution of a program proceeds one basic block or segment at a time. At the start of execution, the segment driver is called with a parameter indicating which basic block or segment to execute. Before executing a basic block/segment, the segment driver decides whether or not to optimize. If optimization is to be performed, it decides which basic blocks/segments to optimize and how much, and calls the appropriate optimizers. Then it executes the basic block/segment. If the executable code is interpretive, a subroutine call is made on the interpreter; otherwise a subroutine call is made on the machine language representation of the basic block or segment.

During the execution of the code, there may be a call on a subprogram; these calls may be nested. To execute the subprogram, the segment driver is called recursively. Execution of the subprogram proceeds as just described, but any calls on the interpreter must be recursive, for the interpreter may have made the subprogram call.

Execution of the basic block or segment is terminated by a branch instruction to another basic block or segment. If the branch occurs in a

basic block or is external to a segment, control is returned to the segment driver by a subroutine return which passes back the destination of the branch instruction. Branches internal to a segment are performed directly, while a return from a subprogram causes an exit from the segment driver.

This entire process, depicted in Figure 2.1, is repeated until an instruction that terminates the program is executed.

The system we implemented and will describe employs the incremental dynamic optimization strategy. The segment driver for such a system operates as just described, except now the optimization count determines when to optimize.

When a basic block/segment is to be executed, the segment driver decrements its associated optimization count. If the result is negative, the next optimization in sequence is performed; this calculates a new optimization count for the basic block/segment. Then the basic block/segment is executed as previously described. The modified flowchart of the segment driver is given in Figure 2.2.

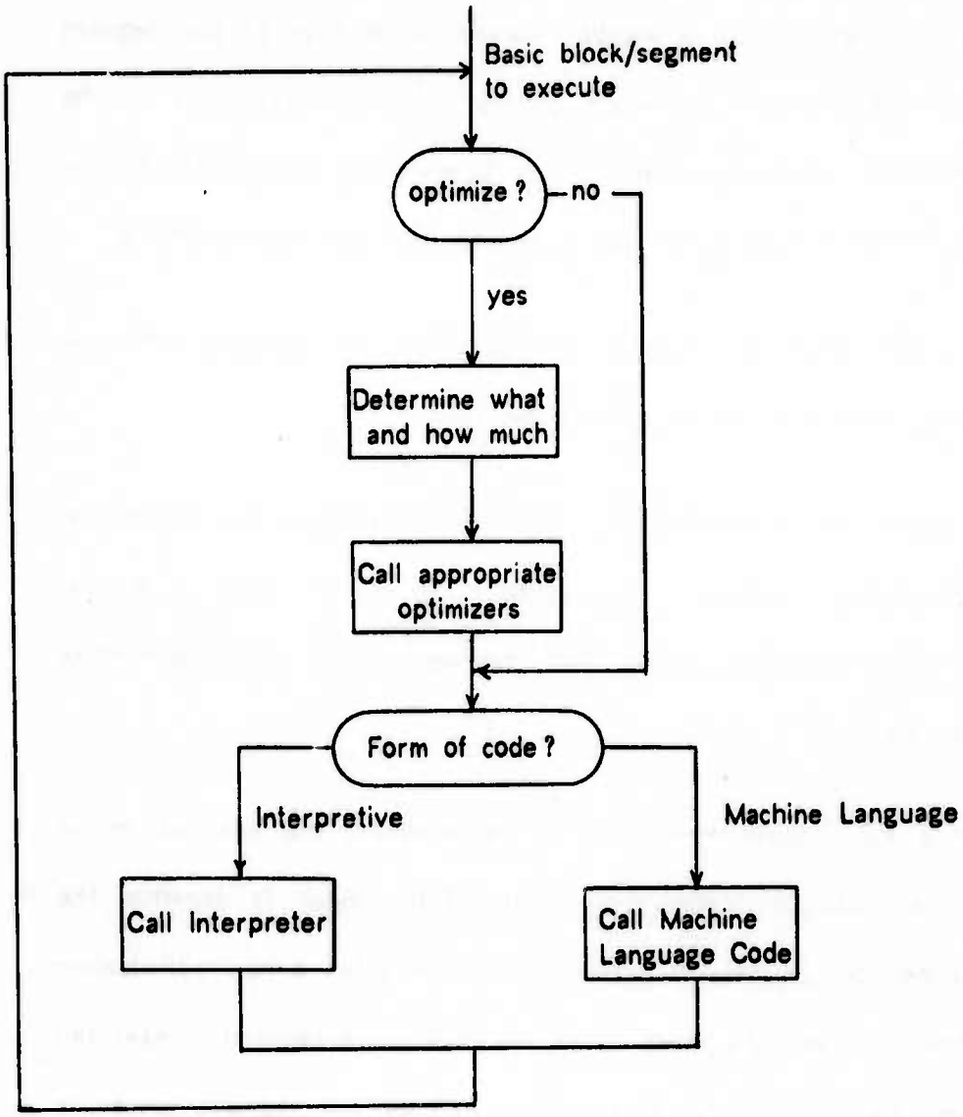


Figure 2.1: The Execution Cycle for a General Segment Driver

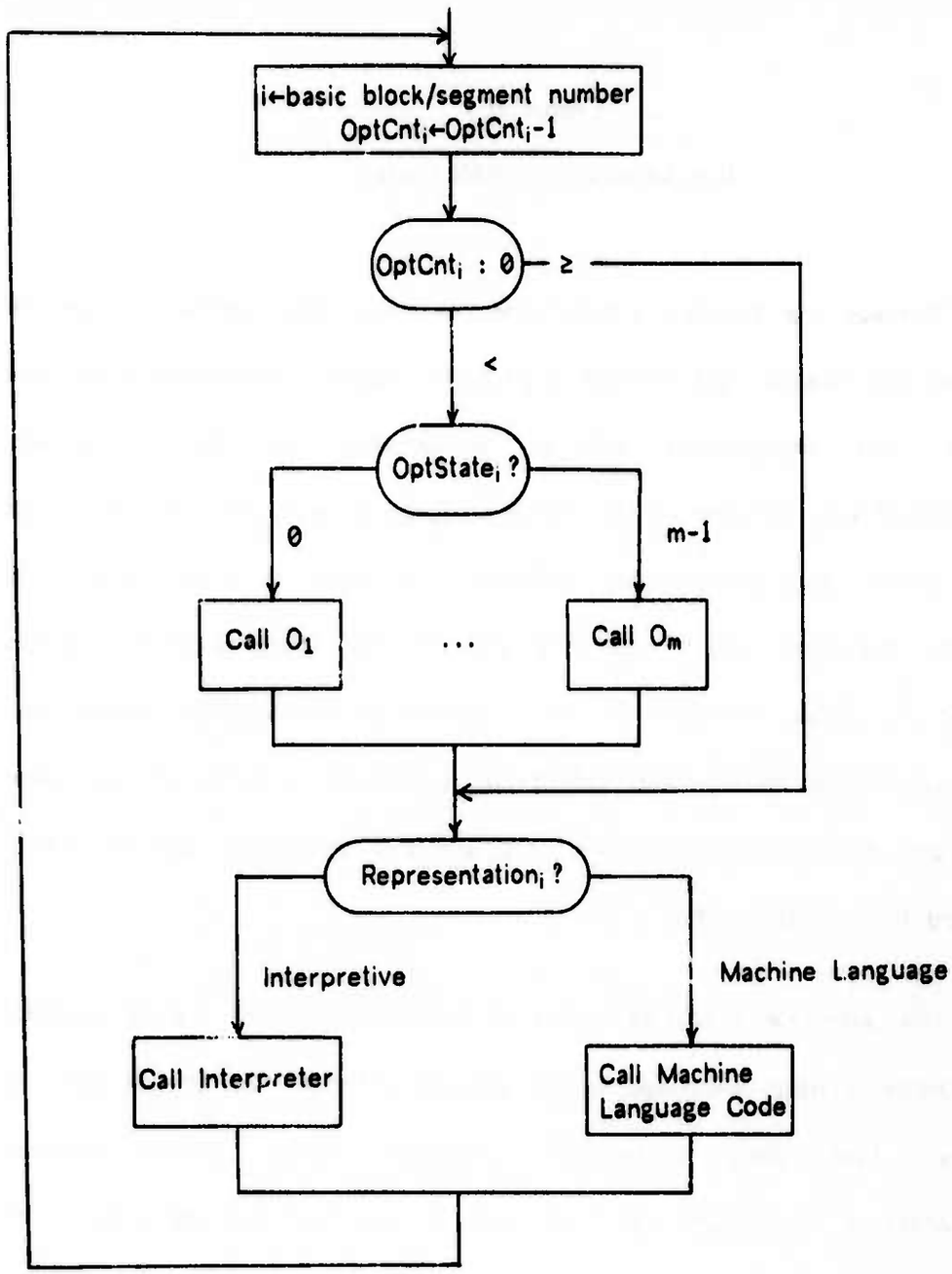


Figure 2.2: Execution Cycle of Segment Driver for Incremental Dynamic Optimization

Chaper III

The Adaptive FORTRAN System

Whereas the previous chapter was concerned with adaptive systems in general, this chapter will describe a particular adaptive FORTRAN system; this system was implemented and its performance has been measured. FORTRAN-IV was selected as the source language because: 1) it is one of the most widely used programming languages and hence is a rich source of example programs and comparisons with existing systems; 2) it contains enough interesting constructs to give credibility to the validation results; and 3) many of the compile-time optimization algorithms currently in use were developed for FORTRAN compilers; they are well understood and are easily adapted for use at run-time.

The Adaptive FORTRAN system is based on the incremental dynamic optimization scheme described in the previous chapter (see Section 2.5). It employs four basic optimizations (constant folding, fusion, common subexpression elimination, and code motion), and has two generators for translating the internal representation of the source code (quadruples) to machine language. The chapter is divided into two major sections. The first section will describe the organization of the system (i.e., the different system modules and the function of each), design criteria and implementation details. The bulk of this section may be bypassed on a first reading without loss of continuity. However, it is suggested that the introduction to Section 3.1.2 on

the system's structure be read and Figure 3.1 be looked at. The second section is important, for it describes the final system and how it was arrived at through an evolutionary chain of systems. The latter discussion also includes a presentation of the final system's optimization states and their associated optimization counts. We will defer a discussion on the performance of the system until the next chapter.

3.1 The System's Design and Implementation Specifications

So that the Adpative Fortran system may be clearly understood and duplicated, a detailed description of its design and implementation is presented.

3.1.1 The Adaptive FORTRAN Language

In order to demonstrate that our technique is workable and valid, it is not necessary to strictly adhere to the formal definition of FORTRAN-IV or to implement the entire language. We assume the reader is familiar with FORTRAN-IV. Instead of describing the complete subset, we therefore list all the features in FORTRAN-IV that were altered, extended or deleted. The following extensions and alterations were made:

1) allow an arbitrary number of dimensions for arrays,

2) allow multiple assignment statements,

3) allow the use of real as well as integer control variables in DO statements,

4) allow the use of parameters as initial, incrementation, and terminal values in DO statements,

5) allow the use of negative increments in DO statements,

6) allow the use of expressions in output lists,

7) perform automatic conversion of real to integer type for subscripts, in relational expressions and in DO statements,

8) include exclusive OR and equivalence as logical operators.

The following features were deleted:

1) the use of embedded blanks in identifiers,

2) the use of double precision and complex arithmetic as types,

3) usage of the computed GO TO statement,

4) usage of the PAUSE statement,

5) usage of auxilliary and unformatted I/O statements,

6) the use of the DO-implied specification in I/O lists,

7) usage of the DATA statement,

8) usage of the EQUIVALENCE specification statement,

9) the use of statement functions,

10) the requirement that symbolic names which identify statement types or operators may not be reserved words,

11) the ability to compile program units separately.

These modifications were made because they simplified the experiments without affecting their results.

3.1.2 Structure of the System

The process of running a FORTRAN program is broken down into three major phases: 1) the compilation of FORTRAN source code to relocatable quads; 2) the loading of the relocatable quads to absolute quads; and 3) the execution of the program. Execution of the program is controlled by a supervisor known as the segment driver (see Section 2.6) which conditionally invokes an optimizer before allowing a basic block or segment to execute. Execution of a basic block or segment is performed either by: 1) the interpreter which interprets the quads; or 2) the machine language equivalent of the quads, called as a subroutine. When optimization is performed, the optimizer performs transformations on the quads and creates a machine language segment by calling appropriate generators. The decision of when and what to optimize is controlled by the optimization count and optimization state associated with the basic block/segment. However, performance of the system depends on the optimization states selected and how the associated optimization counts are determined.

The ability to change these two optimization control parameters easily and thereby produce different systems whose performance can be studied was a major design criterion applied in the design and implementation of the optimizers. Each optimizer is designed as a self-contained module which accepts as an input parameter the basic block/segment to be optimized. It either deduces all the information it needs to perform the optimization or

obtains it from the segment table (a common data structure accessible to all optimizers). An optimizer module consists of two subroutines: one which performs the optimization algorithm and a second which makes all the control decisions associated with the optimization. Typical control decisions are: performing possible setups, calling the optimization algorithm, changing the optimization state and optimization count for the basic block/segment, calling machine language generators, optionally outputting statistics about the optimization (e.g., processing time, number of quads manipulated or modified) and performing any cleanup functions. This modular construction isolates those few parts of the system that must be modified in order to produce a different experimental system.

The structure of the system is shown in Figure 3.1.

The bulk of the system was written in BLISS-10 [Bli71], a systems programming language for the DEC PDP-10. Those portions of the system written in machine language were the segment driver (hand optimized to minimize overhead) and the run-time FORTRAN support package (the mathematical routines, I/O package, etc.) borrowed from the PDP-10 FORTRAN system with slight modifications.

The entire system is loaded at once into approximately 50K 36 bit words. This is not necessary; the three phases could be overlaid (and would be in a production quality system). Again, this does not affect the validity of the results.

3.1.2.1 The Compiler

The first phase in running a FORTRAN program is the translation of the FORTRAN source text into the internal form manipulated by the optimizers. The internal form selected is a quadruple, or quad for short, which consists of an operation, OP, two operands, A1 and A2, and a result temporary, T. A quad has the form:

(OP, A1, A2, T).

The compiler is one pass and compiles relocatable quads directly to core. It occupies approximately 9K of core and compiles at the rate of nearly 9,000 cards/minute. Its structure is modeled after an ALGOL compiler written by the author and fellow colleagues [Hay71].

A secondary function of the compiler is to partition the program into basic blocks. Code is compiled into a basic block until the occurrence of one of several conditions in the source text, at which time the basic block is terminated and another one started. The conditions are:

1) a labeled statement (except a FORMAT statement),

2) a subroutine or function call (except for a library function or output subroutine call, since they produce no side effects, i.e., they do not change the value of a variable),

3) a "call exit" (e.g., STOP, RETURN) or END statement,

4) statement(s) which cause the generation of a consecutive sequence of conditional transfer operations possibly terminated by an unconditional transfer (see Appendix A, Section A.2.3, specifically the arithmetic and logical IF).

5) a GO TO statement,

or 6) a READ statement.

During compilation each result generated in a basic block is associated with a unique temporary location. This is to facilitate the translation of quads to machine language and the optimization of the basic block. (Since these are intermediate results pertinent only to the basic block in which they occur, a different basic block may utilize the same temporary locations. See Appendix B, Section B.1.)

3.1.2.2 The Loader

After all program units have been compiled, the relocatable quads are immediately loaded by a loader (see Section 2.1) if the program contains no errors. The loader occupies less than 0.5K of core, and is very fast (all the relocatable quads are in core).

The primary function of the loader is to load the quads into absolute core locations; this requires changing relative locations to absolute and 'back patching' address fields. Before the loading process can commence, the loader must first determine how the program is to be laid out in core memory, i.e., it must determine the starting absolute address for each relocation base (the unit of storage into which code is compiled). All compiled addresses are relative to one of several relocation bases: sequential instruction storage, out-of-sequence instruction storage, own storage,

temporary storage, non-COMMON variable storage, blank COMMON storage, labeled COMMON storage, library storage, and segment table storage.

The second function of the loader is to build the **segment table**, which is crucial for the adaptive process. Each entry in this table contains all the information about a basic block that is needed by the optimizers. A single entry in the table consists of the following information fields:

1) QUADREP: The address of the basic block's first quad. Initialization occurs at load time (the compiler generates the starting address of each basic block under the segment table relocation base).

2) CURREP: The absolute address of the current representation of the basic block. When the block is executed, this address determines how it is executed. The initial value is the address of the quad interpreter; when the basic block's quads are translated to machine language, the value is the starting address of the machine language.

3) SEGNO: The segment number to which the basic block belongs when the basic block is fused into a segment. Initially it is equal to the basic block number. After fusion, it is the block number of the segment's entry block. Thus, the identity of an embedded segment is lost. In the case of non-homogeneous fusion, embedded segments are remembered by saving on a list the block number of the first and last block of the segment. This list is associated with the covering segment by saving a pointer to it in another field appended to the segment table.

4) OPTCNT: The basic block/segment's optimization count. This field is decremented by the segment driver each time the basic block/segment is executed by the segment driver. When it goes negative, the basic block/segment is

optimized according to the OPTSTATE field.

5) OPTSTATE: The optimization state of the basic block/segment. This field determines which optimization is to be performed next on the basic block/segment when the OPTCNT field goes negative.

6) PREDPT: A pointer to the first item in the linked list of immediate predecessors for the basic block. This list contains the block number of all basic blocks that are immediate predecessors of the block in increasing order.

7) LASTPRED: A pointer to the last item in the basic block's immediate predecessor list.

8) QB: The address of the first quad branch instruction in the basic block. This field is used when it is necessary to move the machine language for the basic block and the quad branch instructions must consequently be retranslated to machine language.

9) MLB: The starting address of the machine language translation of the quad branch instruction(s) in the basic block. When the machine language for a basic block is moved, only those machine language instructions from CURREP to this address need be moved.

10) AENTRY: The machine language address of the alternate entry point to the segment's entry block. The segment's invariant quads are affixed to the start of the segment's entry block (see Section 3.1.3.3). When the segment is translated to machine language, the CURREP field points to the first machine language instruction of the segment's entry block, i.e., to the invariant code. But the invariant code need only be executed once, hence any internal branch to the segment's entry block need only go to the alternate entry point. When the quads for the

segment entry block are translated to machine language, the AENTRY field is set so all subsequent quads of the segment that branch to the entry block will be translated to branch to the address specified by it.

After the program is loaded, the loader initializes the segment table. The fields are initialized to the following values:

1) CURREP is set to the address of the interpreter,

2) SEGNO is set to the basic block's block number which is identical to the entry's placement in the segment table (numbers starting at 1). Thus the block number is used as an index into the table.

3) AENTRY is set to zero,

4) OPTCNT is set to a constant which determines how long the basic block is to be interpreted (see Section 3.2),

5) OPTSTATE is set to zero (see Section 3.2 for the possible values this field may attain and their meanings),

6) PREDPT and LASTPRED are set as the quads of each basic block are scanned in the generation of the immediate predecessor lists. The quads of a basic block are scanned backwards, since in order to determine immediate predecessors it is necessary to examine only the branch instructions which terminate the basic block.

7) QB is set when the immediate predecessors are being generated, for the first branch instruction in the basic block is the last branch instruction scanned (see (6) above).

After the segment table has been initialized and the immediate predecessors generated, the loader examines the loop structure of the program. Based on the loop structure it changes the OPTSTATE and OPTCNT fields of certain basic blocks. This part of the loader is dependent entirely on the incremental dynamic optimization scheme employed. Therefore we

defer discussing the details of this loop structure analysis until Section 3.2.

The loader terminates by passing control to the segment driver and specifying to it the first basic block in the main program to be executed.

3.1.2.3 The Execution Phase

Execution of the program is controlled by the segment driver (see Section 2.6). The main loop of the segment driver consists of two machine language instructions: one decrements the OPTCNT field for the basic block/segment being executed and tests if the count has gone negative; the other calls the interpreter or machine language segment as a subroutine. If the optimization count goes negative, the basic block/segment is optimized according to the OPTSTATE field before being executed. Execution of the basic block/segment is terminated by a branch instruction that transfers control out ot the basic block/segment. The branch behaves as a subroutine return so control is returned to the segment driver, which executes the next basic block/segment specified by the branch. Thus the overhead incurred is two machine language instructions in the segment driver plus the number of instructions to effect the branch. If the branch is being interpreted the overhead is approximately 12 machine language instructions; if it is in machine language, the overhead is two instructions.

It should be pointed out that not only does the segment driver call the interpreter, but that it is possible for the interpreter to call the segment

driver. Therefore, both routines must be recursive. The latter situtation arises when the interpreter calls a subprogram unit. The reasons for the recursive call is that the segment driver controls the execution and optimization of the program, i.e., execution and optimization proceeds one basic block/segment at a time. Calling a subprogram unit is the only case in which the execution of a basic block/segment is interrupted while other basic blocks/segments are executed (and possibly optimized). Centralizing the control of execution and optimization in the segment driver provides a clean interface between the interpreter, the optimizers, and the program sections in machine language, and enables the control to be changed easily so different systems can be constructed and experimented with. The segment driver can also be directly called recursively if the basic block/segment is in machine language and contains a call on a subprogram unit. The reason is the same as for the indirect recursive call, but now the quad calling the subprogram has been translated to equivalent machine language, i.e., code which is identical to that executed by the interpreter. A subprogram return is the only branch out of a basic block/segment in which control is not to be passed back to the segment driver, but back to the point where the segment driver was called recursively. To effect the exit from the segment driver, the subprogram return passes back a block number of zero which the segment driver executes. The CURREP field for block zero in the segment table points to an alternate entry point in the segment driver which contains the exit code. Thus the same control mechanism is used to effect all

branches out of a basic block/segment.

Program execution thus consists of executing, via the segment driver, one basic block/segment at a time with optimizations intermixed. We now turn our attention to the various optimizations implemented.

3.1.3 The Optimizations

Adaptive FORTRAN uses four machine independent optimizations: constant folding, non-homogeneous fusion, common subexpression elimination and code motion, and a host of machine dependent optimizations. There are a number of reasons why these optimizations were selected over other possibilities. First, these optimizations are the most commonly used ones. Second, they allow us to construct systems similar in characteristics to existing compilers against which it is possible to compare the Adaptive FORTRAN system (see Chapter 4). Third, to show the flexibility of the system, we wanted to include optimizations that applied both to basic blocks and segments. Finally, we wanted to include enough optimizations to prove the technique was not only feasible, but that the system could perform at least as well as current compiler systems.

There are two machine language generators which apply various machine dependent optimizations. The first is the "dumb" code generator, which performs straight forward translation of quads to machine language. It is used when individual basic blocks are being optimized. The second

machine language generator is the "fair" code generator, which is considerably more sophisticated. It utilizes information gathered from the translation of previous quads and in certain cases combines consecutive quads in order to generate more efficient machine language. It is used to generate machine language for optimized segments.

Optimization is either at the basic block level (fusion and/or the "dumb" code generator), or the segment level (common subexpression elimination or code motion in combination with the "fair" code generator). Regardless of which is used, the net effect is the creation of machine language from the basic block/segment's quads. For a segment, the machine language for each basic block must occupy consecutive core for execution purposes. Therefore, it is built piecemeal by appending the machine language for successive basic blocks in the segment.

If an optimization has no effect on a basic block and the proper machine language exists, all machine language instructions except those for the branches (which terminate the basic block) can be moved because they are position independent. The branches must be retranslated. (The instructions which must be moved can be determined from the CURREP and MQB fields for the basic block in the segment table. The QB field specifies where the quads are located for the branches that must be retranslated.)

If the machine language for the basic block does not exist, the proper generator is called and it will compile the machine language directly to the

end of the machine language segment being formed. Since the segment is built piecemeal, there is a problem with forward branches to blocks not yet processed. This is handled by chaining the branch instructions together and then patching them when the block is processed.

The translation (or retranslation) of quad branches is handled specially in order to minimize the overhead for inter-block transfers. The problem is determining the correct machine language to be generated for the branch, i.e., whether any should be generated at all, and if so, whether the machine language should perform the branch directly or go through the segment driver. The correct decision depends on whether the branch is internal or external to a segment (see Section 2.3). For an external branch, the machine language goes through the segment driver so the destination will be optimized further. In the case of an internal branch, either: 1) no machine language is generated if the branch is unconditional and the destination is the next basic block; or 2) the machine language performs the branch directly via the CURREP/AENTRY field in the segment table because optimization of the destination is controlled by its segment entry block. After the final optimization has been performed on a segment, a branch in one of its basic blocks to the entry block is considered to be internal so it will be performed directly.

Whether the branch is external, internal via CURREP or internal via AENTRY is encoded in the quad (see Appendix A, Section A.3, specifically the

BTY tag). The current value of the tag aids in determining the correct machine language to be generated and saves having to regenerate the information. It is updated whenever the branch is translated or retranslated to machine language in order to reflect the (possible) change in status of its containing basic block brought about by the application of an optimization.

We turn now to a brief description of each optimizer in order to give a clear understanding of how they work (and their limitations).

3.1.3.1 Fusion

When a basic block has been executed enough times, it is fused into a segment having the properties given in Section 2.2. The fusion process consists of two parts: the logical determination of the segment containing the basic block and the physical creation of the machine language segment.

The logical segment is determined by the fusion algorithm which utilizes the immediate predecessor lists and the SEGNO field in the segment table (for bypassing the examination of immediate predecessor lists of basic blocks already fused into a segment). As a consequence of the algorithm, a segment consists of a set of consecutively numbered blocks, i.e., a segment is a contiguous section of the segment table. After the segment is formed, the SEGNO fields of all basic blocks in the segment are changed to be the block number of the segment entry block.

The physical machine language segment is created by the control section of the fusion module. Adaptive FORTRAN uses non-homogeneous fusion. If the machine language for a basic block already exists, it is used; otherwise the basic block's quads are translated to "dumb" code.

Finally, the fusion optimizer determines the new optimization state and optimization count for the new segment (see Section 3.2 for the precise values used and how the optimization count is determined).

3.1.3.2 Common Subexpression Elimination (CSE)

The CSE optimizer eliminates common subexpressions from a basic block. The optimizer is not applied to the segment taken as a whole, but to each basic block contained in the segment whose optimization state indicates CSE has not yet been performed (embedded segments may already have had CSE performed on them).

The optimization is performed on the quad representation of the basic block. All modifications are made directly to the quads; temporary locations may therefore be used more than once (in the original compiled code each result of a basic block was assigned a unique temporary) and no-operation (NOP) instructions placed where common subexpressions have been eliminated.

The CSE algorithm makes two passes over the basic block's quads. The prepass searches for replacement operations on simple variables and, using

this information, determines the limit for each quad, i.e., the first quad which changes the value of one of its arguments. The limit of a quad puts a bound on the quads that must be searched when searching for a common subexpression.

The second pass over the quads searches for common subexpressions, i.e., for two quads that have identical operation codes and input arguments. This search is accomplished by scanning forward to the limit of the quad. If an identical quad is found, it is replaced by a NOP and the usage of the result temporary for the NOP'ed quad is searched for (it must occur in a quad that occurs after the NOP'ed quad but before the limit of the quad) and changed to be the result temporary of the identical quad.

Since the optimizer has already collected information on the location of each quad involving a replacement operation, these quads are searched for pairs from which the intermediate temporary can be eliminated, i.e., for quad sequences of the form:

(OP,V,E,T) or (OP,E,V,T)
(=,T,,V)

which can be collapsed to:

(OP,V,E,V) or (OP,E,V,V)

where OP is a binary or unary operator, V is a simple variable, E is a result temporary or simple variable and T is a result temporary. This collapsing enables the machine language generators to produce more efficient code, and saves them from having to regenerate the same information in order to

perform the collapsing themselves.

Since each basic block of the segment is processed separately, the machine language segment is generated simultaneously. After CSE is performed on the basic block its quads are translated to machine language using the "fair" code generator. If the optimization state of the basic block indicates CSE has already been performed, then the "fair" code already exists and it is simply moved in a manner identical to that previously described.

The entire process is controlled by the control section of the module which also determines the new optimization state for each basic block and the new optimization count for the segment.

3.1.3.3 Code Motion (CM)

Code motion eliminates invariant quads in a segment. A quad is invariant if the arguments of its operation are invariant within the segment. Invariant quads are replaced by a NOP and are collected together in a new basic block called the invariant code block. This block is logically appended to the segment's entry block. It is not physically appended to the entry block for implementation reasons: 1) certain optimizations assume (for efficiency purposes) that the quads for each basic block occupy contiguous memory locations, and to append the invariant quads would require moving quads to make room and updating the segment table; and 2) it provides a cleaner solution to the problem of how to translate to machine language

internal branches to the entry block, for these branches should not be to the invariant code block.

To logically connect the invariant code block with the segment entry block, the invariant block is terminated by a special branch quad of the form: (JUMP,EB,QREP,QBR). EB is the block number of the entry block, QREP is the QUADREP field from the segment table for the entry block and is known as the alternate entry point to the segment, and QBR is the QB field from the segment table for the entry block. These three pieces of information constitute what is needed to move the entry block's machine language or to generate its machine language. The invariant code block is made the new segment entry block by changing in the segment table for the old entry block:

1) the QUADREP field to the address of the first quad in the invariant block,

and 2) the QB field to the address of the special branch quad which terminates the invariant code block.

See Appendix B, Section B.1 for an example of an invariant code block, the machine language generated for it, and the entry block associated with it (especially the code generated for a branch to the alternate entry point).

The CM algorithm first makes sure CSE has been performed on each basic block in the segment (no machine language is generated). Then in order to find the invariant quads, it makes two passes over each basic block in the segment. In the first pass, it constructs a list of all variables or

indirect results that are not invariant. Using this list, it then searches each basic block for invariant quads; however it processes only the invariant code block for embedded segments which have already had CM applied to them.

Let the quad being processed by CM be of the form:

Q1: (OP,A1,A2,T1)

If the quad is invariant, i.e., its arguments A1 and A2 are invariant, then how it is processed depends on whether or not it is in an invariant code block and if it already exists in the new invariant code block.

Suppose Q1 does not already exist in the new invariant block. If Q1 is not in an invariant code block, then the quad (OP,A1,A2,T3) is added to the new invariant code block, where T3 is a new unique temporary (using a new unique temporary is necessary since basic blocks share the same temporary locations). Then Q1 is replaced by the quad (REPL,T3,,T1), read T1←T3, if T1 must be in memory (see Appendix A, Section A.3, specifically the SR tag); otherwise with a NOP. All occurrences of T1 occurring after Q1 in the basic block are replaced by T3. If Q1 is in an invariant code block, T1 is a unique temporary, so the quad (OP,A1,A2,T1) is inserted into the new invariant code block and Q1 replaced with a NOP.

If on the other hand Q1 is already in the new invariant code block, then it is a common subexpression that is invariant in more than one basic block (recall CSE is performed only on individual basic blocks of a segment, not on the segment taken as a whole). Let the common subexpression in the

new invariant code block be of the form: (OP,A1,A2,T2). Then if Q1 is in an invariant code block, T1 is a unique temporary and it suffices to insert the quad (REPL,T2,,T1) in the new invariant block and replace Q1 with a NOP. If Q1 is not in an invariant code block, then Q1 is replaced by the quad (REPL,T2,,T1) if T1 must be in memory; otherwise with a NOP. All occurrences of T1 occurring after Q1 in the basic block are replaced by T2.

The net effect of the algorithm is to cause quads to "bubble" to the outermost segment (loop) of which they are invariant.

The control section of the CM module invokes the CM algorithm and then generates the machine language segment. For those basic blocks in which invariant code was removed and for the entry block to which invariant code was appended, machine language is regenerated using the "fair" code generator.. The ("fair") machine language for the remaining basic blocks already exists and is moved in a manner identical to that previously described.

As is the case for the other optimizers, the final function performed by the module's control section is to determine the new optimization state for the basic blocks of the segment and the new optimization count for the segment (see Section 3.2 for the exact values used).

3.1.3.4 The "Dumb" Code Machine Language Generator

The "dumb" code generator operates on basic blocks, and is invoked when it is no longer advantageous to interpret a basic block or when a basic block is fused into a segment and is still in interpretive code form. In keeping with the philosophy of incremental dynamic optimization (i.e., gradual optimization of a section of code), it is a fairly straightforward translation of quads to machine language, and employs a trivial register allocation scheme and some of the less sophisticated machine dependent optimizations.

The register allocation algorithm uses four working registers that it assigns on a round robin basis. When a register is needed, the algorithm checks if the register after the last register assigned is free. If not, it generates code to store the register in its associated temporary. A temporary is associated with a register when it is the result temporary of a quad. Once the temporary is used as an argument, it is disassociated from the register because each result generated in a basic block uses a unique temporary. Variables are not associated with a register. For those operations (e.g., integer division) that require two consecutive registers, a single register is first located in the manner just described. Then if the next higher register is in use, code is generated to store it.

Generation of the machine language is table driven. For each possible quad op-code, there is a control word which specifies what machine language is to be generated and how. The control word is broken into a number of

fields: the type of the operation, the register specification of the arguments, the register specification of the result, whether the quad has embedded machine language, the number of machine language instructions to be generated, a pointer to the machine language instructions, an indicator for CSE and CM eligiblity, and a switch to differentiate between conditional and unconditional branches. Encoded in the address fields of the machine language instructions are integers specifying which argument of the quad to use.

The generator does not make a fine distinction between the op-codes and therefore does not generate specialized code to handle each situation, but instead classifies the op-codes into four groups. The operation type field in the control word specifies which class the op-code belongs in: commutative binary, non-commutative binary, unary and all others. The quad is processed according to this operation type.

As a result of this classification of operations, the number of machine dependent optimizations that can be performed is limited. These optimizations consist of:

1) the use of "immediate" instructions for literal constants (constants less than 18 bits),

2) the use of indexing for indirect results,

and 3) recognizing for a binary or unary operation the arguments are in a register and utilizing that register in forming the result.

The translation of quad branches to machine language is a special case; processing is as previously described. If machine language is generated for an internal branch, it performs the branch directly through the CURREP field in the segment table.

The "dumb" code generator occupies approximately 1.5K of core. It is fairly fast, taking on the average of 550μs to process a quad. The generated code executes approximately 9 time faster than it takes to interpret the equivalent quads.

3.1.3.5 The "Fair" Code Machine Language Generator

The "fair" code generator is applied to segments, one basic block at a time. It is invoked after the CSE or CM optimizer has been applied to the segment.

Generation of the machine language involves a thorough case analysis of the variables for each operation in order that the most appropriate PDP-10 instructions can be used. The PDP-10 instruction set is quite extensive; most instructions have a basic form plus a number of variants. To utilize the complete instruction set and therefore generate the "ultimate" machine language would involve an unreasonable amount of effort, certainly more than necessary to validate our approach. Therefore, the operations were ranked according to frequency of usage with a corresponding detailed analysis.

The case analysis for the binary and unary operators is based on the mode of the arguments involved. The possible modes and a brief reason for each are:

1) MEM: argument in memory. This mode handles variables that are in memory and results that have to be stored in order to free a register.

2) REG: argument in a register. This mode is for retaining variables across replacement statements and intermediate results.

3) NUM: argument is a number. This mode permits the processing of literal constants (constants less than 18 bits) and constant folding.

4) REG+NUM: argument is the result of adding the contents of a register to a number. This mode delays the generation of the addition so that if the argument is used as an indirect result, indexing can be used (the NUMber is the address field and the REGister the indexing register).

Using the mode of an argument as a coordinate label and an argument to label each dimension, a code array is constructed for each binary and unary operation (cf. [Gri71]). Each element of the array contains the code to be generated for that particular case. For the binary operators, there are 16 possible cases, while for the unary operations there are only four cases corresponding to the four modes.

Most of the cases are subdivided into subcases. The correct subcase is selected according to information stored in either of two data structures: the temp table or the register table. There is one entry in the temp table

for each temporary used in the basic block; each entry consists of six fields:

1) Mode of temporary result:
   a) MEM: result has been stored into memory
   b) NUM: result is a folded number
   c) REG: result is in a register
   d) REG+NUM: result is a register plus a number

2) Register associated with temporary, i.e., the register the result occupies.

3) **Range** of temporary, i.e., the address of the last quad that uses the temporary. When the quad is processed, the temporary is disassociated from the register it is in.

4) **Neg-bit**, which indicates the negative of the temporary is required. This bit permits the generation of negation instructions to be delayed, and therefore allows multiple negations to cancel one another or special instructions to be generated (e.g., load/store negative, subtract instead of add, etc.).

5) Information field, which contains the address of a constant or the value of a folded constant or literal.

6) Number indicator, which identifies the number in the information field; either:
   a) the number is not a result of folding and the information field contains the address of the constant,
   b) the number is the result of folding and the information field contains the value of the constant (which is not a literal). Whenever an instruction is generated that uses this constant, storage must be assigned for it and initialized to its value,
   c) the information field contains the value of a literal (folded or otherwise).

The register table contains one entry for each working register; each entry consists of eight fields:

1) Mode of the register:
   a) register has no associated temporary.
   b) register has an associated temporary whose mode="REG".
   c) register has an associated temporary whose mode="REG+NUM".

2) The use of the register, which indicates how many temporaries with mode="REG+NUM" are associated with the register.

3) The variable counter, which indicates how many variables are associated with the register. This allows variables to be retained in registers after a replacement operation and thereby possibly avoids the generation of a reduandant load instruction.

4) The address of the associated temporary with mode="REG".

5) Fields for specifying the address of variables associated with the register (there may be up to four).

The information contained in these two data structures permits the following machine dependent optimizations:

1) constant folding,

2) use of special instructions to set memory to 0 or -1,

3) use of shift instructions for multiplication or division by powers of 2,

4) delaying negation operators to make use of load/store negative instructions, permiting the usage of complement instructions for an operation, or deleting successive negation operations,

5) use of "immediate" instructions for operations involving literal constants as arguments,

6) use of indexing for indirect results (subscripting),

7) performing operations directly to memory, e.g., incrementation or decrementation by a literal constant or for quads of the form: (OP,V,E,V) where V is a simple variable and E is a simple variable or result,

8) performing operations both to memory and a register simultaneously, e.g., for quads of the form: (OP,V,E,V).

These optimizations are but a small sample of the optimizations that could be performed if we were to exploit the full instruction set of the PDP-10. They were selected because they have a high payoff for the effort invested.

The operations were broken down into three classes with varying degrees of analysis applied. The most detailed analysis is performed on the integer arithmetic operators: binary +,-,*,/ and unary minus, since integer arithmetic is required in frequently used language constructs (e.g., for counter variables that control the number of times a loop is executed or for subscript variables that reference array elements). For the binary operators there are three code matrices, each designed to handle quads of a specific form (see Appendix C for the '+' code matrix). The three forms are:

(BINOP,E1,E2,T)
(BINOP,V,E,V)
(BINOP,E,V,V)

where BINOP is one of the binary operators +,-,*,/; E, E1 and E2 are either simple variables, results or indirect results; V is a simple variable; and T is a temporary. There are two code vectors for unary minus. One handles quads of the form (-,E,,T) while the other is for quads of the form (-,V,,V).

The next class of operations consists of the floating point arithmetic and logical operators. There is one code matrix to handle the binary floating point arithmetic and logical operators. Parameters to the code matrix are the machine language instructions for the operator that handle the different cases. There are separate code vectors for unary floating point negation and logical not.

The final class of operations includes all the remaining operations. The entire analysis is performed by one subroutine, and the production of the machine language is table driven as it was for the "dumb" code generator. However, the analysis is more involved due to the different modes the arguments may attain.

Most of the analysis is independent of the operation being performed, but there are two types of operations that require special processing. The first special case involves the branch operations. The analysis for determining the correct machine language is identical to that previously described except that now there is another case to consider if CM was applied to the segment. This involves external branches to the segment entry block. If an invariant code block was appended to the segment entry block by CM, then these external branches must be to the alternate entry point and not to the invariant block. For these external branches, the machine language performs the branch directly through the AENTRY field in the segment table instead of the CURREP field which is the address of the

invariant block's machine language. If CM did not create an invariant block, then external branches to the entry block are direct through the CURREP field, not through the segment driver, since CM is the last optimization that can be applied.

The other special case is for relational operators. This is the only other case besides CSE in which a sequence of quads is examined in order to produce more efficient machine language. The FORTRAN construct being optimized is the logical IF of the form:

IF(E1 ROP E2)S

where ROP is a relational operator and S is a statement. The quads generated for this construct can be found in Appendix A, Sections A.2.1 and A.2.3, but it is basically the pair:

(ROP,E1,E2,T)
(OP,T, ,)

that is being combined to eliminate the intermediate logical result T, where OP is either BF, BT, STOPT, EXTST or EXTFT (see Appendix A, Table A.1).

Germane to the generation of efficient optimized code is the effective use of the registers. Whereas the quads operate strictly on temporaries, the generated machine language instructions use registers, and it is up to the machine language generator to control how the registers are utilized. One means of using the registers effectively is for the generator to remember what variables and results reside in which registers so that those registers can be used to form further results, thereby avoiding redundant load

operations. Thus, for example, the fact a binary operator is commutative is recognized so the result is formed in the register occupied by one of the arguments (if either argument is already in a register) or if a replacement statement of a result into a simple variable is generated, the variable is associated with the register so it will not be reloaded if used later.

The other means for controlling the use of registers resides in the register allocation algorithm, which is invoked whenever a register is needed. The algorithm assigns the least recently used of the 10 working registers. When a register is needed, the registers (actually the register table) are searched starting with the last register assigned. First a search for a register not in use is made. If this fails (i.e., all registers are in use), then a search for a register with no associated temporary is made, starting with the first. This search is effectively for a register that only has variables associated with it. If this fails, then it is necessary to store a register containing a result. A search is made for a register with an associated temporary having the minimum number of associated variables. If this fails, then all registers have a mode of "REG+NUM", so the register with the smallest number of associated temporaries is selected. Code is generated to perform the addition and store the result.

There are cases (e.g., integer division) when two consecutive registers are needed. There is another form of the register allocation algorithm that is identical to the one just described, but which searches for two consecutive

registers possessing the same properties. If both registers do not have the same property, another search is made to find at least one with the desired property. Only if this fails is a search continued for two consecutive registers with another identical property.

The "fair" code generator requires approximately 10K of core. It takes on the average twice as long to process a quad as the "dumb" code generator, i.e., approximately 1200μs. However, code generated for a basic block runs on the average twice as fast as the code generated by the "dumb" code generator.

3.2 The System's Optimization States and Their Associated Optimization Counts

Performance of the system depends on: 1) how the fusion optimizer forms a machine language segment (homogeneous versus non-homogeneous); 2) what optimizations are applied (individually or in combination) and in what order; and 3) the optimization counts. The modularity of the system and the isolation of the code that controls the behavior of the optimizers provide the ability to change the adaptive strategy easily and thereby produce operationally different systems.

Approximately 15 different systems were constructed and tested before the final form was determined. As each system was tested, more insight into the dynamic optimization process was gained. The performance of each successive system was analyzed and this led to experiments involving

variations in the control functions. The final system is a result of this evolutionary process.

The first systems tested used homogeneous fusion; all basic blocks in a segment simultaneously attain the same optimization state, which is the maximum optimization state of any basic block contained in the new segment. By studying the performance curves it became apparent that performance was not satisfactory for small execution times. It was deduced that during the early stages of a programs execution, too much optimization was being applied too soon. The problem then was to obtain satisfactory performance for small execution times without degrading performance for medium-to-large execution times.

The first attempt to defer the optimization process was to change the optimization counts and keep the optimization states fixed. The optimizations and their order or application were: translation of basic blocks to "dumb" code, homogeneous fusion, CSE and CM. This approach did not prove to be sufficient mainly because embedded segments tend to attain a high optimization state and thereby cause the covering segment to be optimized too fast.

The next set of systems used non-homogeneous fusion as described in Section 3.1.3.1. The results were better than that achieved using homogeneous fusion but still not satisfactory, for while it improved performance for small execution times, it degraded performance for large

execution times. The reason was that the optimization state of embedded segments became frozen. To compensate for this, all embedded segments were advanced to their next optimization state before an optimization (CSE or CM) was performed. This improved the performance for large execution times, but degraded the performance for medium execution times. Therefore, another approach was needed for controlling the optimization rate of embedded segments.

We decided to perform a pre-analysis on the loop structure of the program before execution started, and to change the optimization state and optimization count of certain basic blocks from their initialized values according to their depth of nesting in the loop structure. We first attempted to increase the optimization rate of innermost loops since they are executed the most frequently and therefore should be optimized first. We hoped the additional optimization time would be negligible compared to the savings in execution time. Only innermost loops consisting of two or fewer basic blocks were considered. Three different ways in which the initial optimization of these innermost segments could be allowed to proceed were considered:

1) The translation of a basic block's quads to "dumb" code if the basic block is executed more than once.

2) Fusion to "dumb" code if any basic block in a segment is executed more than once.

and 3) Total optimization of a segment if any basic block in it is executed more than once.

The results were encouraging, with the second of the three approaches being

the most promising. However, performance for medium execution times was still being degraded because the optimization rates of the non-innermost segments were the same. Therefore, the loop analyzer was modified to recognize innermost and outermost segments, thereby partitioning the segments into three classes. The outermost segment's optimization count for the last optimization state was made smaller than that for other embedded segments because its execution rate is slower than that for these other segments and therefore it should not be executed the same number of times before being totally optimized. This final modification produced the most favorable results.

The optimization states for the final system on whose performance we shall report in the next chapter were as follows:

0: translate the interpretive code for the basic block to "dumb" machine language.

1: perform a non-homogeneous fusion of the basic block into a segment. Basic blocks in interpretive code are translated to "dumb" machine language; blocks in machine language are moved as is with their branches retranslated.

2: perform code motion on the segment. Before the optimization is performed, the CSE algorithm is performed on all basic blocks in the segment. After the optimization, the quads of each basic block are translated to "fair" machine language.

Note that CSE is not a separate optimization, but is combined with CM. The reason was that the time to generate the machine language segment using the "fair" code generator is appreciably larger than the time to perform

the CSE or CM algorithm. Therefore, the combined time to perform CSE and CM separately is much greater than the time to perform the combination, because the machine language segment must be generated twice.

The optimization counts associated with each of the optimization states depend on the loop structure of the program. In the loop classifications that follow, the triplet (C0,C1,C2) represents the optimization counts for optimization states 0, 1 and 2 respectively:

1) innermost segments (i.e., loops): (0,1,50). Thus innermost segments are fused into "dumb" machine language if executed once, then totally optimized. The loop analyzer initializes the optimization state and count for basic blocks belonging to an innermost segment to 1.

2) outermost segments: (6,15,n) where n = 10 if the length of the segment (in basic blocks) is ≤ 10; otherwise 2*length.

3) other segments: (6,15,200).

4) entire subprogram: (6,15,n) where n is the same as in 2). However, CM is not performed as the last optimization; instead CSE is applied to all the basic blocks in the subprogram. It makes no sense to remove invariant quads out of a subprogram because the entire subprogram is always executed when called. Thus the entire subprogram is considered a segment and processed as any other segment with respect to optimization. What constitutes an outermost loop inside a subprogram depends on whether the subprogram is called from within a loop. It is assumed that this is always the case for it this assumption is not made, experiments indicate that system performance is degraded.

Since the third optimization count is determined prior to program execution, it must be saved. This is accomplished by appending another field

to the segment table.

Finally, an explanation of how the optimization counts were determined is in order. The final values given above are based on findings obtained by experimenting with a system that employed homogeneous fusion. Corresponding to that system's four optimization states, there were four optimization counts: OP0, OP1, OP2, and OP3. Initially, basic blocks and segments were treated uniformly. The optimization counts for those in the same optimization state were assigned a constant value that did not depend on any attribute of the basic block or segment. The values selected and the reasons were:

1) OP0=10. OP0 controls when a basic block is translated to "dumb" code. Since translation time is 550μs/quad and interpretation time is approximately 25μs/quad, an upper bound on the number of times the basic block should be interpreted before being translated is 550/25=22 times. However, this calculation does not take into consideration the execution time of the new representation. If the basic block continues to be executed, it would pay to translate sooner because its execution time will be less. Thus, a fraction of 22 was selected, viz., approximately 1/2.

2) OP1=15. Determining OP1 is harder, because the amount of time required to perform the homogeneous fusion algorithm is a function of the length of the segment and cannot be determined a priori. Therefore, the value choosen is based on the fact that fusion should be performed as soon as possible, but not before the benifits of being in "dumb" code could be felt, i.e., the effort required to translate quads to "dumb" code should not be wasted.

3) OP2=35. Since CSE produces code that is at least twice as fast as that produced by fusion to "dumb" code, a value was choosen which is approximately twice OP1.

4) OP3=70. Because the benifits of going from CSE-"fair" code to CM-"fair" code are not as great as going from fused "dumb" code to CSE-"fair" code, a value was choosen that could delay performing CM for a reasonable amount of time.

In an attempt to improve performance for small execution times these optimization counts were varied slightly, with no appreciable results. Since OP3 was thought to be the most critical factor, other functions for determining it were tried based on the length of the segment measured in either number of quads or number of basic blocks, e.g., taking the natural logarithm or a constant multiple. The most promising was taking a constant multiple (2) of the length measured in basic blocks.

It became apparent that constant optimization counts were not sufficient to significantly improve performance. Further improvements were made by changing to non-homogeneous fusion, combining CSE with CM, and not treating segments uniformly, but classifying them according to their level of nesting in a loop structure. This necessitated adjusting the optimization counts accordingly. The values choosen are given above; the reasons are:

1) innermost segments: C1=1 for there is no reason to perform any optimization if the segment is not executed at least once. To totally optimize the segment after it is executed once results in too much optimization being applied too soon. Therefore, total optimization is delayed, and since CSE was combined with CM, C2=OP1+OP2=50.

2) outermost segments and entire subprograms: C0=OP0=10 proved to be too high a value, while C1=5 was minimal. Therefore, C0=6 was choosen. C1=OP1=15 for the same reasons given for OP1. C2 is a multiple of the length of the segment measured in basic blocks because this function was experimentally the most promising for

determining when the final optimization should be performed.

3) other segments: C2=200 because OP3 was considered to be too small for a segment that is in a loop structure at least three levels deep. Based on an analysis of how many times such a loop could be executed in such a loop structure, 200 seemed a reasonable choice.

It is unfortunate that the optimization counts were determined heuristically and a more theoretical basis was not found. But the excellent performance results presented in the next chapter speak for themselves.

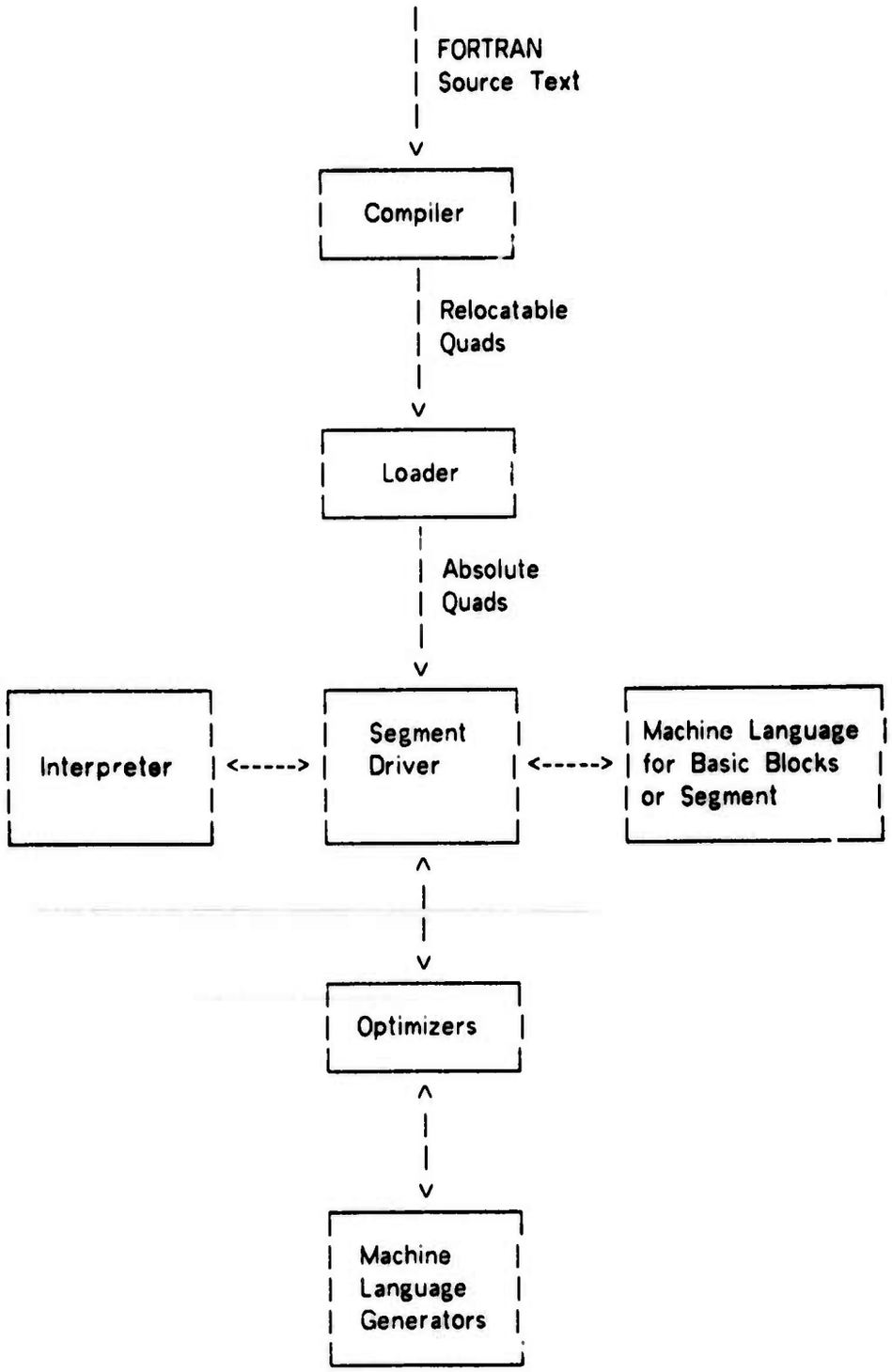


Figure 3.1: Structural Organization of the Adaptive FORTRAN System

Chapter IV

Validation and Experimental Results

In this chapter we present the experimental evidence which demonstrates that dynamic optimization is a workable and valid technique. The demonstration strategy consists of implementing the Adaptive FORTRAN system described in the previous chapter, and measuring its performance on an appropriate program mix.

In order to evaluate Adaptive FORTRAN's performance measurements, it is necessary to compare the results with those obtained by running the same set of test programs under other types of FORTRAN compilers, viz., WATFIV, FORTRAN-IV G and FORTRAN-IV H. To do this for various machines would not, unfortunately, provide a meaningful comparison due to the differences in the machines and their compilers. For the comparisons to be meaningful, the same compiler, optimizers, machine language generators and object machine should be used.

Therefore, the approach taken is to transform the Adaptive FORTRAN system into systems that resemble those three real compilers. It is against those three compilers, plus the DEC PDP-10 FORTRAN compiler (F40), that the Adaptive FORTRAN system is compared. We will present the results of running the test programs under the five different systems in tabular and graphical form, and discuss their implications.

4.1 Comparative Compiler Systems

In order to have a basis for evaluating how dynamic optimization compares to current compilers, it is necessary to run a number of test programs under different compiler systems and to compare the performance measurements. For FORTRAN, there are three well known classes of FORTRAN compilers that might be used for comparison purposes:

1) WATFIV: a one pass compiler that compiles directly to core. It is very fast and the code produced is fairly decent.

2) FORTRAN-IV G: usually a multi-pass compiler that produces a relocatable object module that must be loaded by a standard system loader. It compiles relatively fast and generates code that is better than that produced by WATFIV. Optimization is at the basic block level. The generated code is comparable to that produced by CSE and the "fair" code generator.

F40, the PDP-10 FORTRAN compiler, can be classified as a G-type compiler except that optimization is at the statement level. It compiles relocatable code to a disk file. A standard system loader creates in core an absolute load module from one or more relocatable object modules. This load module can be saved as a file on disk and called for execution.

3) FORTRAN-IV H: a multi-pass optimizing compiler that optimizes the entire program at compile-time. The output is a relocatable object module. The compiler is usually a few times slower than FORTRAN-IV G. However, the object code is usually two or three times faster than that produced by FORTRAN-IV G (see Low[69]).

There are a number of practical problems in making the comparisons. First, not many computers have all three compilers available. Second, the differences in characteristics of various computers (e.g., speed, word size, and instruction set) must be taken into account; this complicates the comparisons. Finally, the differences between the compilers themselves, e.g., the parsing and code generation techniques employed, the type of machine language generated, and the run-time support package must be taken into account.

The ideal situation would be to have all the compiler systems run on the same machine and to use the same compiling techniques, optimizers, machine language generators and run-time support package. The construction of these compiler systems was considered to be too large an undertaking. Therefore, a more expedient approach was taken in which the Adaptive FORTRAN system (AF) was transformed to resemble each of the other three compilers. It was easy to make the necessary changes because of the way AF was constructed (see Sections 3.1.2 and 3.1.3). The main discrepancy between the transformed systems and the actual compilers lies not in the type of code produced, but in the way it is produced. Each transformed system uses the Adaptive FORTRAN compiler to translate FORTRAN source text into quads. After loading the quads, but before starting execution, the quads are translated to the machine language form that most resembles the code produced by the compiler being emulated. We feel this discrepancy in no way alters the validity of the test results, since the use of a consistent approach does not bias the results.

The three transformed systems and the manner in which they produce code are:

1) AFW: resembles WATFIV. The entire program is translated to "dumb" code before execution starts. A true WATFIV compiler does not produce quads, but compiles machine language directly. Like WATFIV, AFW compiles in one pass directly to core. But whereas WATFIV's machine language is absolute and requires some patching before execution starts, AFW produces relocatable quads which must be loaded. Therefore, the compiler-loader phases of a WATFIV compiler probably would be slightly faster than those for AFW.

2) AFG: resembles FORTRAN-IV G. CSE is performed on all the program's basic blocks and then the entire program is translated to "fair" code before the start of execution. A FORTRAN-IV G compiler usually produces relocatable machine language directly to a disk file. The absolute load module is created from the relocatable object modules by a standard system loader. Since these modules are on disk, load-time should be greater than that for AFG which uses a specialized in core loader (see Section 3.1.2.2). The compile-time for FORTRAN-IV G should be comparable to the combined time required by AFG to compile and load the program and perform the translation of the quads to machine language. Therefore, the compiler-loader phases of AFG should be slightly faster than that for FORTRAN-IV G.

3) AFH: resembles FORTRAN-IV H. All optimizations are applied to the entire program which is then translated to "fair" code before the start of execution. CSE is first performed on each basic block in the entire program. Then the segments are formed via fusion starting with the innermost ones and working outwards. The list of segments and their order of processing is given to the system, not deduced by it. As each segment is formed, CM is performed on it. After all segments are formed, the entire program is translated to "fair" code. A true FORTRAN-IV H

compiler also produces an internal form such as quads which its optimizers process (cf. [Low69]). The compiler-loader phases of AFH should be slightly faster than that for FORTRAN-IV H for the same reasons given above for AFG.

These three compiler systems form the basis against which AF is compared. The performance of each system was measured by running the same set of test programs under each.

4.2 The Test Programs

In order to draw meaningful conclusions about the performance of AF, it is necessary to run a number of test programs under it that have different characteristics. Care must be exercised in selecting the test programs to avoid biasing the results. For any compiler system, it is always possible to construct a program that makes it look miserable or one that makes it look good. To ensure that the test programs are representative of the type of programs written in the real world, both the published literature and students were used as sources.

A number of criteria were used to select the test programs from the potential candidates; they were designed to test if the usage of AF is restrictive. The main criterion was to select programs with differing loop structures, e.g., a different number of loops, loop lengths (measured in basic blocks) and loop nestings. The reason was that we wanted to test AF's performance both on those class of programs it was designed for (i.e.,

programs for which 5% of the code accounts for 50% of the execution time) and on those that do not fall into this classification.

Second, we wanted programs that have parameter(s) that can be varied to control their execution time. This allows us to study the performance of AF for small, medium and large execution times, and determine if performance is a function of the execution time.

Finally, we wanted programs that were compute bound in order to do a worse case analysis. I/O bound programs were not selected because the I/O handlers are not part of the user's program and cannot therefore be optimized by the system, and if the program performs any I/O, the I/O time is a constant for a fixed test point regardless of the version of the experimental compiler system being run under. Thus, the analysis of the results is unaltered since it is the difference between the measurements that is relevant when making comparisons.

The four test programs selected (see Appendix B for a listing of the source) and their characteristics are:

1) EE: A student electrical engineering problem.
   a) Control parameters: C2I and C3I, increments that control the accurracy of the results C2 and C3 respectively. For the test runs, C2I was held fixed while C3I was allowed to vary.
   b) Program units: Main program unit only
   c) Number of statements: 51
   d) Number of basic blocks: 9
   e) Number of individual loops: 1
   f) Loop size: 7 basic blocks
   g) Loop nesting: 1 single level

This program is to typify the type of program written by a student. It was obtained from an EE student [McW72].

2) SIEVE2: A prime number generator [Cha67].
   a) Control parameter: K, the number of primes to be generated
   b) Program units: Main program unit only
   c) Number of statements: 86
   d) Number of basic blocks: 27
   e) Number of individual loops: 7
   f) Loop sizes(in basic blocks):
      1,2(2),4,5(2),25†
   g) Loop nesting:
      1 single level
      5 double level
      1 triple level

This algorithm is a modification of Chartres' algorithm in that it generates the first K primes instead of all the primes ≤ M.

3) LES: A linear equation solver [For67 and Mol72].
   a) Control parameter: N, the number of variables
   b) Program units: Main program unit plus 2 subprogram units
   c) Number of statements: 97
   d) Number of basic blocks: 45
      MAIN: 15
      DECOMP: 20
      SOLVE: 10
   e) Number of individual loops: 13
      MAIN: 4
      DECOMP: 5
      SOLVE: 4
   f) Loop sizes(in basic blocks):
      MAIN: 1,2,4(2)
      DECOMP: 1(2),3,4,18
      SOLVE: 1(2),3(2)

---

† The notation is to be interpreted as follows: for the 7 individual loops, one is of size 1, two of size 2, one of size 4, two of size 5, and one of size 25.

g) Loop nesting:
MAIN: 2 single level
2 double level
DECOMP: 1 single level
3 double level
1 triple level
SOLVE: 2 single level
2 double level

The original algorithm given in the textbook by Forsythe and Moler [For67] consists of two subroutines. However, Moler later published new subroutines that were a modification of, and replacement for, the corresponding original routines [Mol72]. These were the routines used in the program. The test matrices were generated by the program and correspond to Example 3.6 in the book by Gregory and Karney [Gre69] (see Appendix B, Section B.2).

4) QZ: An eigenvalue problem [Mol73].
a) Control parameter: N, the size of the square input matrix
b) Program units: Main program unit plus 9 subprogram units
c) Number of statements: 654
d) Number of basic blocks: 323
MAIN: 9
QZ: 7
QZHES: 66
QZIT: 97
QZVAL: 49
QZVEC: 77
HSH3: 5
HSH2: 5
CHSH2: 5
CDIV: 3
e) Number of individual loops: 51
MAIN: 2
QZHES: 19
QZIT: 12
QZVAL: 4
QZVEC: 14
HSH3, HSH2, CHSH2, CDIV: 0

f) Loop sizes(in basic blocks):
MAIN: 2,5
QZHES: 2(12),3,5,7(2),21,23,32
QZIT: 2(7),3,9,10,44,66
QZVAL: 2(3),42
QZVEC: 2(6),3(2),5,7,18,19(2),48

g) Loop nesting:
MAIN: 1 single level
1 double level
QZHES: 3 single level
7 double level
9 triple level
QZIT: 3 single level
2 double level
7 triple level
QZVAL: 1 single level
3 double level
QZVEC: 3 single level
7 double level
4 triple level

This program was obtained from Stewart and is described, but not given in his paper with Moler [Mol73]. The test matrices are generated by the program and were suggested by Stewart (see Appendix B, Section B.5). This algorithm is interesting in that the intermediate quantities produced by the program may not be the same owing to rounding errors. Consequently, the execution times are theoretically not strictly comparable. However, for practical purposes they are, i.e., the timings depend in a uniform manner on the size of the matrix.

Each of these test programs were run under AF, AFW, AFG and AFH, plus F40, the FORTRAN-IV compiler on the PDP-10. We now present the results of these test runs.

4.3 The Test Results

The performance of each compiler system is measured by obtaining the total run-time for a test program as a function of its control parameter, where total run-time is the sum of compilation time, load time and execution time. The timings were made on a PDP-KA10 computer system with Ampex core having a 1.8μs read/write cycle. In order to obtain accurate timings, it is necessary to run the compiler systems with no load on the computer system, for timings are sensitive to the system load. During a test run, the computing environment consisted of the monitor, the I/O handlers and the particular compiler system being tested. Identical computer runs produced the same timings so there is no statistical fluxuation in the results. A 10μs clock was used to make the timings, which are given here in seconds.

The results of the test runs are presented in tabular and graphical form. There are five tables for each program (Tables 4.1-4.4):

1) Compiler and Loader Timing Statistics
   The following statistics are tabulated for F40, AFW, AFG, AFH, and AF:
   a) Compilation time,
   b) Load time,
   c) Total of a) and b),
   d) Optimization time, i.e., that part of the compilation time spent optimizing the program,
   e) The percent of the compilation time spent optimizing, i.e.,
      (optimization time/compilation time)*100.

2) Execution Times

The execution times of the program for F40, AFW, AFG, AFH and AF are tabulated as a function of the control parameter. Data points were taken until the amount of time spent optimizing the program became constant, i.e., until no more optimizations were performed.

3) Total Run-time

The total run-time (compilation time plus load time plus execution time) is tabulated as a function of the control parameter for F40, AFW, AFG, AFH and AF.

4) Total Run-time Ratios

This table indicates the relative speed of AF as compared to each of the compiler systems. The ratio of total run-times is tabulated as a function of the control parameter.

5) AF Optimization Statistics

The following statistics are tabulated as a function of the control parameter:

a) Execution time,
b) Optimization time, i.e., that part of the execution time spent dynamically optimizing the program,
c) The percent of the execution time spent optimizing the program, i.e.,
(optimization time/compilation time)*100.

The third table of total run-times represents the measurements for comparing the performance of each compiler system against AF. In order to compare the systems visually, this table is presented in graphical form for each test program (Figures 4.1-4.4). The coordinates of the graph are the total run-time versus the control parameter. The data for each of the compiler systems is plotted on the same set of axes thereby producing a set of performance curves that can easily be compared.

In order to further demonstrate the effects of dynamic optimization, we constructed another compiler system, AFI, which performs no optimizations, but runs the program interpretively. Table 4.5 shows the results of the initial test points for the test programs QZ and LES. These results are plotted on the corresponding graphs.

Finally, we were interested in studying the behavior of AF for very small execution times because the optimization time then constitutes a large percentage of the execution time. We wanted to see how the fraction of execution time devoted to optimization grows and finally peaks. Refined measurements were made for the test programs QZ and LES, and the results are tabulated in Table 4.6. The results also were used in accurately plotting the initial portion of the corresponding performance curves.

Table 4.1a Compiler and Loader Timing Statistics for EE

	Compilation Time	Load Time	Total	Optimization Time	% of Compilation
F40	3.37	2.25	5.62	----	-----
AFW	.59	.03	.62	.07	11.86
AFG	.77	.03	.80	.25	32.47
AFH	.81	.03	.84	.29	35.80
AF	.52	.03	.55	----	-----

Table 4.1b Execution Times for EE

C3I	F40	AFW	AFG	AFH	AF
.5	41.40	35.13	32.82	30.74	31.04
1.0	20.75	17.70	16.55	15.51	15.81
2.0	10.63	9.09	8.50	7.96	8.27
4.0	5.48	4.69	4.39	4.12	4.42
6.0	3.85	3.28	3.08	2.88	3.19
8.0	3.02	2.58	2.42	2.27	2.57

Table 4.1c Total Run-time for EE

C3I	F40	AFW	AFG	AFH	AF
0.5	47.02	35.75	33.63	31.58	31.59
1.0	26.37	18.32	17.35	16.35	16.36
2.0	16.25	9.71	9.30	8.80	8.82
4.0	11.10	5.31	5.19	4.96	4.97
6.0	9.47	3.90	3.88	3.72	3.74
8.0	8.64	3.20	3.22	3.11	3.12

Table 4.1d Total Run-time Ratios for EE

C3I	F40/AF	AFW/AF	AFG/AF	AFH/AF
0.5	1.49	1.13	1.06	.99
1.0	1.61	1.12	1.06	.99
2.0	1.84	1.00	1.05	.99
4.0	2.23	1.07	1.04	.99
6.0	2.53	1.04	1.04	.99
8.0	2.77	1.03	1.03	.99

Table 4.1e AF Optimization Statistics for EE

C3I	Execution Time	Optimization Time	% of Execution
0.5	31.04	.31	1.00
1.0	15.81	.31	1.96
2.0	8.27	.31	3.75
4.0	4.42	.31	7.01
6.0	3.19	.31	9.72
8.0	2.57	.31	12.06

Table 4.2a Compiler and Loader Timing Statistics for SIEVE2

	Compilation Time	Load Time	Total	Optimization Time	% of Compilation
F40	4.45	2.43	6.88	----	-----
AFW	.77	.17	.94	.11	14.29
AFG	.89	.17	1.06	.23	25.84
AFH	1.03	.17	1.20	.37	30.10
AF	.66	.17	.83	----	-----

Table 4.2b Execution Times for SIEVE2

K	F40	AFW	AFG	AFH	AF
10	.07	.06	.06	.02	.12
20	.07	.07	.06	.02	.15
30	.07	.07	.06	.02	.20
40	.07	.08	.07	.03	.21
50	.07	.08	.07	.03	.47
60	.07	.09	.08	.03	.47
70	.08	.10	.08	.04	.47
80	.10	.11	.09	.04	.48
90	.10	.11	.09	.05	.48
100	.10	.12	.10	.05	.49
200	.18	.23	.16	.12	.55
300	.28	.37	.25	.21	.64
400	.38	.52	.34	.30	.73
500	.52	.70	.44	.41	.83
600	.65	.89	.56	.52	.95
700	.78	1.09	.68	.65	1.07
800	.95	1.29	.80	.77	1.19
900	1.10	1.51	.93	.91	1.33
1000	1.27	1.75	1.07	1.05	1.46

Table 4.2c Total Run-time for SIEVE2

K	F40	AFW	AFG	AFH	AF
10	6.95	1.00	1.12	1.22	.95
20	6.95	1.01	1.12	1.22	.98
30	6.95	1.01	1.12	1.22	1.03
40	6.95	1.02	1.13	1.23	1.04
50	6.95	1.02	1.13	1.23	1.30
60	6.95	1.03	1.14	1.23	1.30
70	6.96	1.04	1.14	1.24	1.30
80	6.98	1.05	1.15	1.24	1.31
90	6.98	1.05	1.15	1.25	1.31
100	6.98	1.06	1.16	1.25	1.32
200	7.06	1.17	1.22	1.32	1.38
300	7.16	1.31	1.31	1.41	1.47
400	7.26	1.46	1.40	1.50	1.56
500	7.40	1.64	1.50	1.61	1.66
600	7.53	1.83	1.62	1.72	1.78
700	7.66	2.03	1.74	1.85	1.90
800	7.83	2.23	1.86	1.97	2.02
900	7.98	2.45	1.99	2.11	2.16
1000	8.15	2.69	2.13	2.25	2.29

Table 4.2d Total Run-time Ratios for SIEVE2

K	F40/AF	AFW/AF	AFG/AF	AFH/AF
10	7.32	1.05	1.18	1.28
20	7.09	1.03	1.14	1.24
30	6.75	.98	1.09	1.18
40	6.68	.98	1.08	1.18
50	5.35	.78	.87	.95
60	5.35	.79	.88	.95
70	5.35	.80	.88	.95
80	5.33	.80	.88	.95
90	5.33	.80	.88	.95
100	5.29	.80	.88	.95
200	5.11	.85	.88	.96
300	4.87	.90	.89	.96
400	4.65	.94	.90	.96
500	4.46	.99	.90	.97
600	4.23	1.03	.91	.97
700	4.03	1.07	.92	.97
800	3.88	1.10	.92	.98
900	3.69	1.13	.92	.98
1000	3.56	1.17	.93	.98

Table 4.2e AF Optimization Statistics for SIEVE2

K	Execution Time	Optimization Time	% of Execution
10	.12	.05	41.67
20	.15	.07	46.67
30	.20	.11	55.00
40	.21	.11	52.38
50	.47	.37	78.72
60	.47	.37	78.72
70	.47	.37	78.72
80	.48	.37	77.08
90	.48	.37	77.08
100	.49	.37	75.51
200	.55	.37	67.27
300	.64	.37	57.81
400	.73	.37	50.68
500	.83	.37	44.58
600	.95	.37	38.95
700	1.07	.37	34.58
800	1.19	.37	31.09
900	1.33	.37	27.82
1000	1.46	.37	25.34

Table 4.3a Compiler and Loader Timing Statistics for LES

	Compilation Time	Load Time	Total	Optimization Time	% of Compilation
F40	6.68	2.20	8.88	----	-----
AFW	.99	.10	1.09	.17	17.17
AFG	1.20	.10	1.30	.38	31.67
AFH	1.30	.10	1.40	.48	36.92
AF	.82	.10	.92	----	-----

Table 4.3b Execution Times for LES

N	F40	AFW	AFG	AFH	AF
5	.08	.09	.10	.02	.25
10	.25	.25	.21	.11	.56
15	.65	.62	.48	.31	.85
20	1.42	1.31	.98	.68	1.36
25	2.62	2.42	1.78	1.27	2.04
30	4.37	4.04	2.94	2.12	2.90
35	6.82	6.29	4.56	3.29	4.14
40	10.07	9.25	6.69	4.83	5.68
45	14.08	13.04	9.41	6.79	7.65
50	19.17	17.75	12.79	9.21	10.08
55	25.32	23.48	16.91	12.16	13.03
60	32.68	30.32	21.83	15.68	16.70
65	41.33	38.39	27.62	19.81	20.84
70	51.38	47.78	34.38	24.62	25.66

Table 4.3c Total Run-time for LES

N	F40	AFW	AFG	AFH	AF
5	8.96	1.18	1.40	1.52	1.17
10	9.13	1.34	1.51	1.61	1.48
15	9.53	1.71	1.78	1.81	1.77
20	10.30	2.40	2.28	2.18	2.28
25	11.50	3.51	3.08	2.77	2.96
30	13.25	5.13	4.24	3.62	3.82
35	15.70	7.38	5.86	4.79	5.06
40	18.95	10.34	7.99	6.33	6.60
45	22.96	14.13	10.71	8.29	8.57
50	28.05	18.84	14.09	10.71	11.00
55	34.20	24.57	18.21	13.66	13.95
60	41.56	31.41	23.13	17.18	17.62
65	50.21	39.48	28.92	21.31	21.76
70	60.26	48.87	35.68	26.12	26.58

Table 4.3d Total Run-time Ratios for LES

N	F40/AF	AFW/AF	AFG/AF	AFH/AF
5	7.66	1.01	1.20	1.30
10	6.17	.91	1.02	1.09
15	5.38	.97	1.01	1.02
20	4.42	1.05	1.00	.96
25	3.89	1.19	1.04	.94
30	3.47	1.34	1.11	.95
35	3.10	1.46	1.16	.95
40	2.87	1.57	1.21	.96
45	2.68	1.65	1.25	.97
50	2.55	1.71	1.28	.97
55	2.45	1.76	1.31	.98
60	2.36	1.78	1.31	.98
65	2.31	1.81	1.33	.98
70	2.27	1.84	1.34	.98

Table 4.3e AF Optimization Statistics for LES

N	Execution Time	Optimization Time	% of Execution
5	.25	.09	36.00
10	.56	.29	51.79
15	.85	.34	40.00
20	1.36	.45	33.09
25	2.04	.54	26.47
30	2.90	.54	18.62
35	4.14	.61	14.73
40	5.68	.61	10.74
45	7.65	.61	7.97
50	10.08	.61	6.05
55	13.03	.61	4.68
60	16.70	.77	4.61
65	20.84	.77	3.69
70	25.66	.77	3.00

Table 4.4a Compiler and Loader Timing Statistics for QZ

	Compilation Time	Load Time	Total	Optimization Time	% of Compilation
F40	67.00	3.83	70.83	----	-----
AFW	10.12	.58	10.70	1.65	16.30
AFG	13.58	.58	14.16	5.11	37.63
AFH	15.16	.58	15.74	6.69	44.13
AF	8.47	.58	9.05	----	-----

Table 4.4b Execution Times for QZ

N	F40	AFW	AFG	AFH	AF
5	.45	.62	.52	.74	2.29
10	2.70	3.02	2.03	2.04	4.81
15	7.42	7.85	4.94	4.46	7.63
20	15.42	15.89	9.69	8.34	12.02
25	30.17	30.70	18.33	15.39	20.32
30	48.13	48.43	28.63	23.65	23.63
35	73.62	73.57	43.12	35.22	40.34
40	109.75	109.05	63.38	51.41	56.74
45	149.85	150.91	87.33	70.42	75.81
50	204.45	200.21	115.41	92.58	98.04

Table 4.4c Total Run-time for QZ

N	F40	AFW	AFG	AFH	AF
5	71.28	11.32	14.68	16.48	11.34
10	73.53	13.72	16.19	17.78	13.86
15	78.25	18.55	19.09	20.20	16.68
20	86.25	26.59	23.85	24.08	21.07
25	101.00	41.40	32.49	31.13	29.37
30	118.96	59.13	42.79	39.39	37.68
35	144.45	84.27	57.28	50.96	49.39
40	180.58	119.75	77.54	67.15	65.79
45	220.68	161.61	101.49	86.16	84.86
50	275.28	210.91	129.57	108.32	107.09

Table 4.4d Total Run-time Ratios for QZ

N	F40/AF	AFW/AF	AFG/AF	AFH/AF
5	6.29	.99	1.29	1.45
10	5.31	.98	1.17	1.28
15	4.69	1.11	1.14	1.21
20	4.09	1.26	1.13	1.14
25	3.44	1.41	1.11	1.06
30	3.16	1.57	1.14	1.05
35	2.92	1.71	1.16	1.03
40	2.74	1.82	1.18	1.02
45	2.60	1.90	1.20	1.02
50	2.57	1.97	1.21	1.01

Table 4.4e AF Optimization Statistics for QZ

N	Execution Time	Optimization Time	% of Execution
5	2.29	1.44	62.88
10	4.81	2.48	51.56
15	7.63	2.77	36.30
20	12.02	3.20	26.62
25	20.32	4.39	21.60
30	28.63	4.40	15.37
35	40.34	4.48	11.11
40	56.74	4.65	8.20
45	75.81	4.65	6.13
50	98.04	4.65	4.74

Table 4.5a AFI Timings for QZ

N	Execution Time	Total Run-time
5	2.58	11.63
10	16.69	25.74
15	46.10	55.15

Table 4.5b AFI Timings for LES

N	Execution Time	Total Run-time
5	.28	1.20
10	1.30	2.22
15	3.79	4.71
20	8.40	9.32
25	15.80	16.72

Table 4.6a Refined AF Timings for QZ

N	Execution Time	Optimization Time	% of Execution	Total Run-time
1	.17	.00	0.00	9.22
2	.42	.18	42.86	9.47
3	1.08	.59	54.63	10.13
4	1.72	1.01	58.72	10.77
5	2.29	1.44	62.88	11.34
6	2.74	1.70	62.04	11.79
7	3.16	1.95	61.71	12.21
8	4.09	2.38	58.19	13.14
9	4.26	2.41	56.57	13.31
10	4.81	2.48	51.56	13.86

Table 4.6b Refined AF Timings for LES

N	Execution Time	Optimization Time	% of Execution	Total Run-time
1	.08	.00	0.00	1.00
2	.12	.03	25.00	1.04
3	.17	.06	35.29	1.09
4	.21	.08	38.10	1.13
5	.25	.09	36.00	1.17
6	.35	.18	51.43	1.27
7	.41	.22	53.66	1.33
8	.47	.26	55.32	1.39
9	.49	.26	53.06	1.41
10	.56	.29	51.79	1.48

4.4 Analysis of Test Results

Before analyzing the position of the AF performance curve relative to the curves for AFW, AFG and AFH, we first analyze the relative positions of the AFW, AFG and AFH curves and see if they conform to expectations.

AFW, AFG and AFH differ only in the amount of compile-time optimization they apply to a program, and thus in the efficiency of the machine language they produce. At the start of execution, the AFH curve lies above the AFG curve which in turn lies above the AFW curve. Because of the relative efficiency of the code, the AFW curve eventually will cross the other two, and the AFG curve will cross the AFH curve. These crossover points occur when the difference in compilation time equals the difference in execution time. It is expected then that if a program is run long enough, the AFW curve will lie above the AFG curve which in turn will lie above the AFH curve. If, however, the additional optimization (CM) performed by AFH has no effect, i.e., does not remove any invariant quads from any loop, then the effort is wasted and the AFG curve will lie below the AFH curve.

By checking the tables and figures for each test program, it is seen that the curves follow this behavior pattern. Only for the test program SIEVE2 does the AFG curve lie below the AFH curve. The reason is exactly that given above, viz., CM has no effect. An examination of the optimization results showed that the loops did not contain any invariant quads.

The performance curves for the four test programs indicate that the range of applicability for AFG is very narrow since the crossover point between AFW and AFG is very close to the crossover point for AFG and AFH. One can only conjecture that AFG is not necessary, and that one should run under either AFW or AFH depending on the length of execution.

As for the AF curve, it is expected to initially lie below all the other curves since it performs no initial optimizations. This is indeed the case. If a program is run long enough, the AF curve will asymptotically become parallel to the AFH curve because the executable code becomes identical to that produced by AFH. Indications are that it approaches the AFH curve from above if the time to totally optimize the program exceeds the compile-time optimization time for AFH; otherwise it approaches it from below.

There is a range in which the AF curve might cross over one or more of the other curves and then cross back under. This occurs for small execution times. The width of the range seems to depend on the diversity of the program's loop structure. Consider each test program in terms of increasingly diverse loop structures:

1) EE (see Figure 4.1): A short program having one loop which constitutes most of the program. Since this loop is an innermost loop, the only difference between AF and AFH is that AF first translates the loop to "dumb" code which is executed before being totally optimized. Hence, AF's execution times differ from those for AFH by a constant.

2) SIEVE2 (see Figure 4.2): Like EE, this program contains a main execution loop which constitutes most of the program and gets totally optimized almost immediately.

However, it contains a few embedded loops and is therefore considered an outermost loop. Hence its optimization is more gradual than the loop in EE, but not gradual enough because representations are changed before it is necessary. The spike occurring at the beginning of the AF curve is due to CM having no effect on the outermost loop. This optimization time is wasted and the machine language segment is identical to that produced by CSE.

3) LES (see Figure 4.3): This program consists of three program units, each containing a number of loops. Each subprogram unit contains a doubly nested loop which accounts for most of its execution time. The one program unit gets called only once so the benifits of total optimization are wasted if the program does not run long enough. The other program unit is called repetitively so eventually the double loop and the entire subprogram get totally optimized. The results are excellent, for only AFW is slightly better than AF for small execution times. The initial part of the AF curve is not smooth due to the optimization pertabations which are more apparent for small execution times.

4) QZ (see Figure 4.4): This program contains the most diverse loop structure and consists of 10 program units. Four of the units constitute the main part of the program, and each is called only once. There are a large number of inner loops of 1-2 basic blocks whose early optimization probably contributes to the fact that the AF curve is the best of any test programs. This is also the only case in which the time for total dynamic optimization is smaller than AFH's optimization time. The initial part of the AF curve is not smooth for the same reason stated for LES.

The test results indicate that AF does not outperform all the other systems across the entire spectrum of run-times, but that for a particular program there is a given range in which one compiler system is preferential over any of the others. However, AF is better than any other single compiler system over the spectrum. Thus, we conclude that it is better to

build one compiler system to cover the run-time spectrum than three separate specialized compiler systems, each designed for a different range of the spectrum.

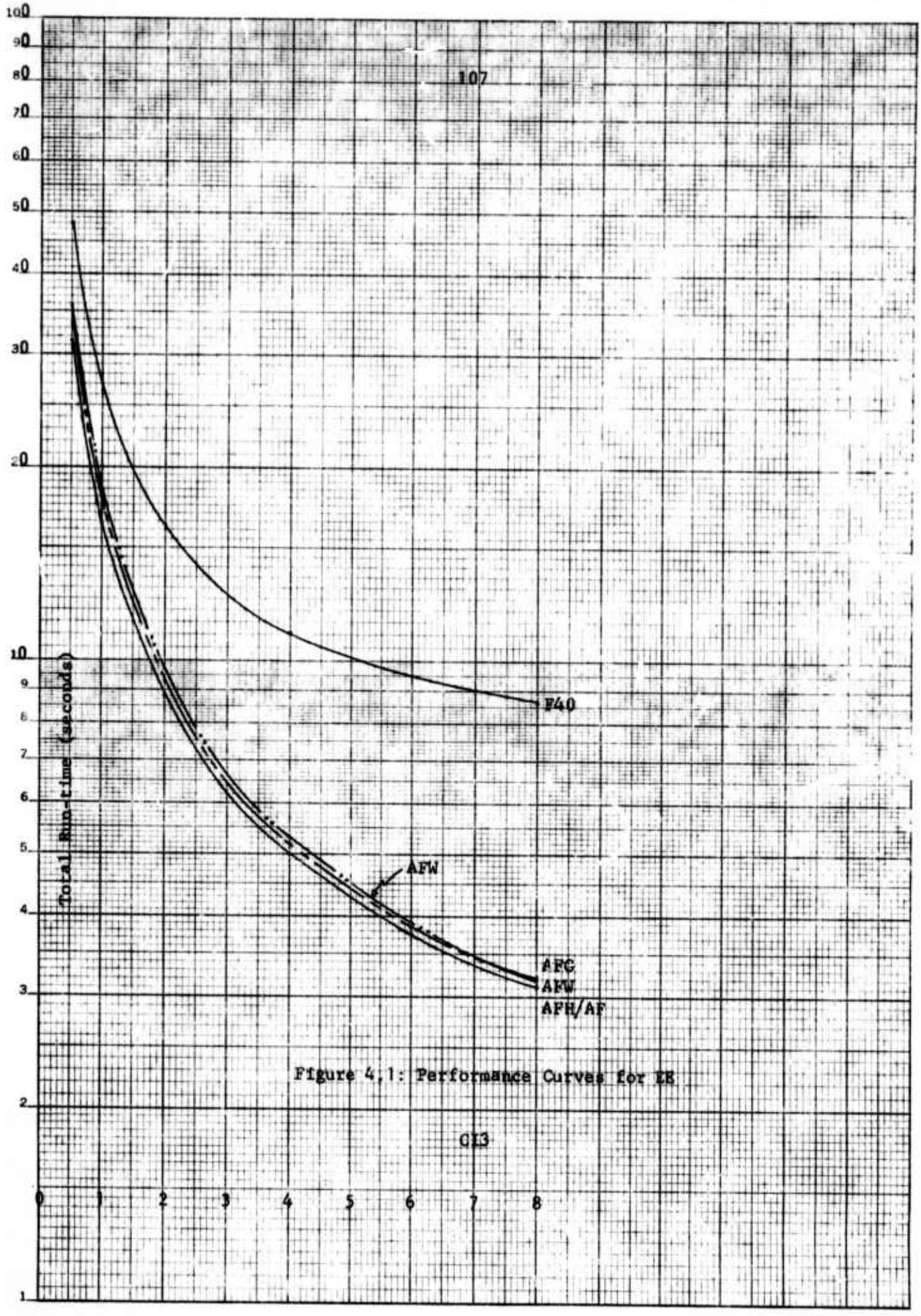


Figure 4,1: Performance Curves for EB

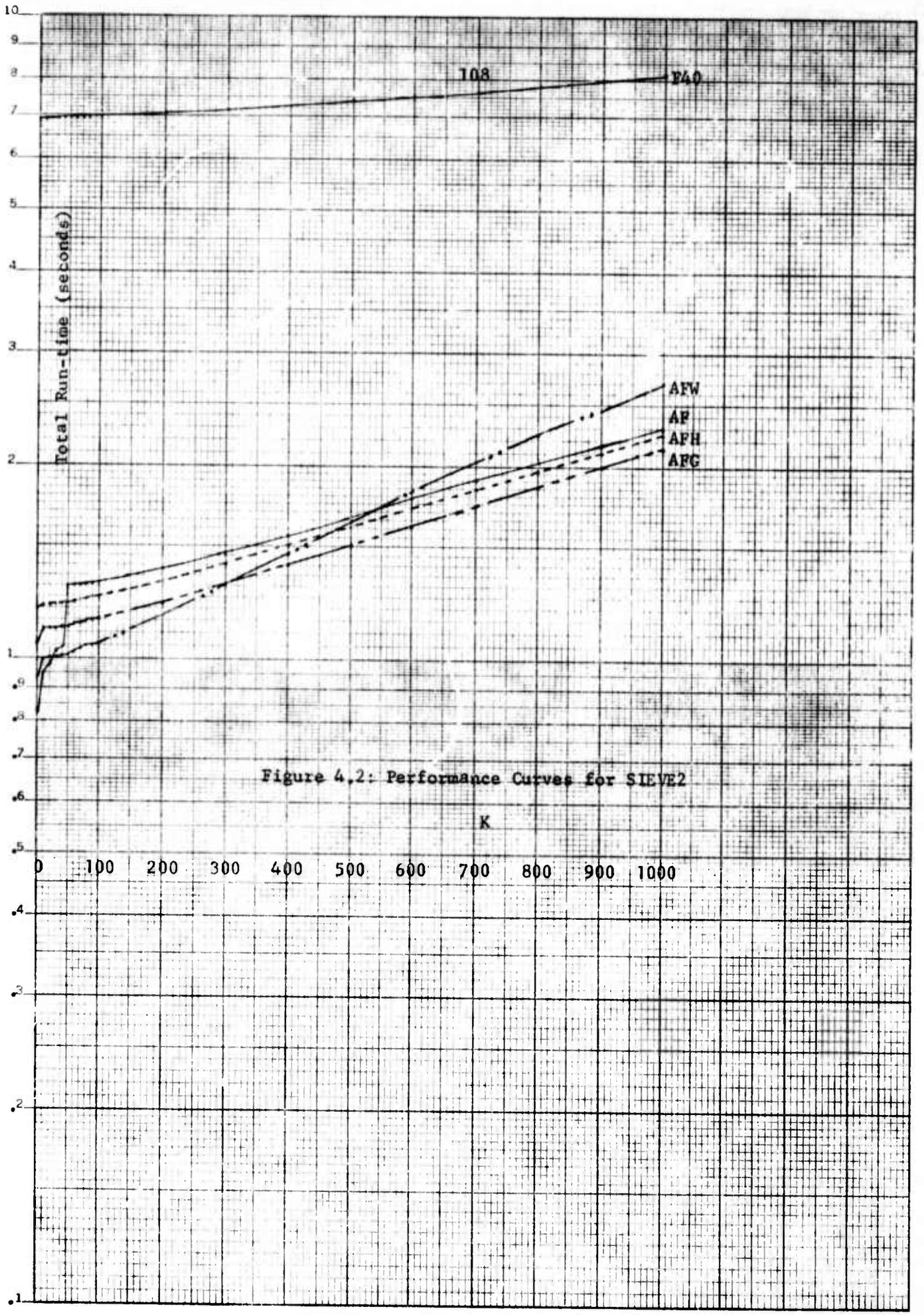


Figure 4.2: Performance Curves for SIEVE2

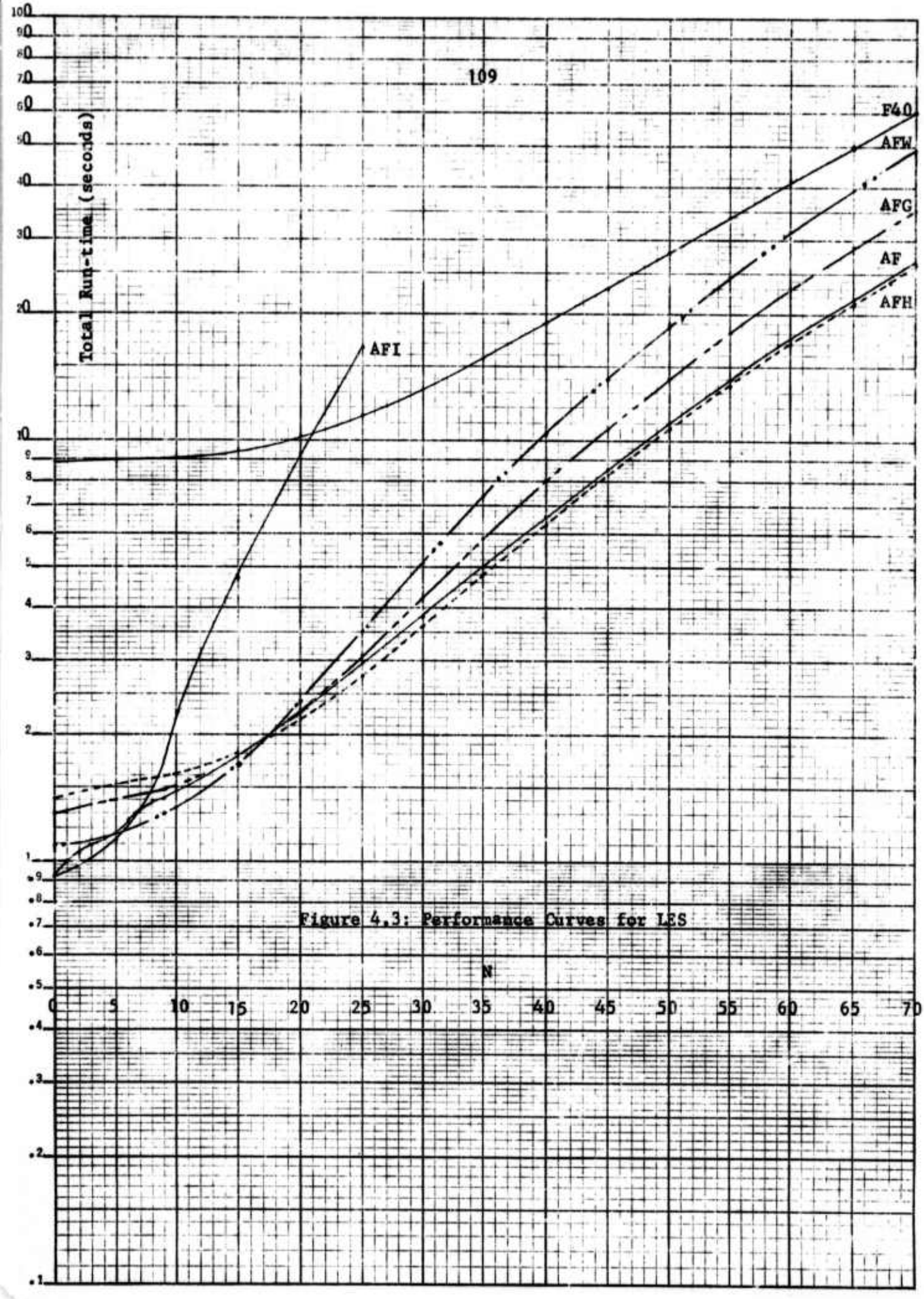


Figure 4.3: Performance Curves for LES

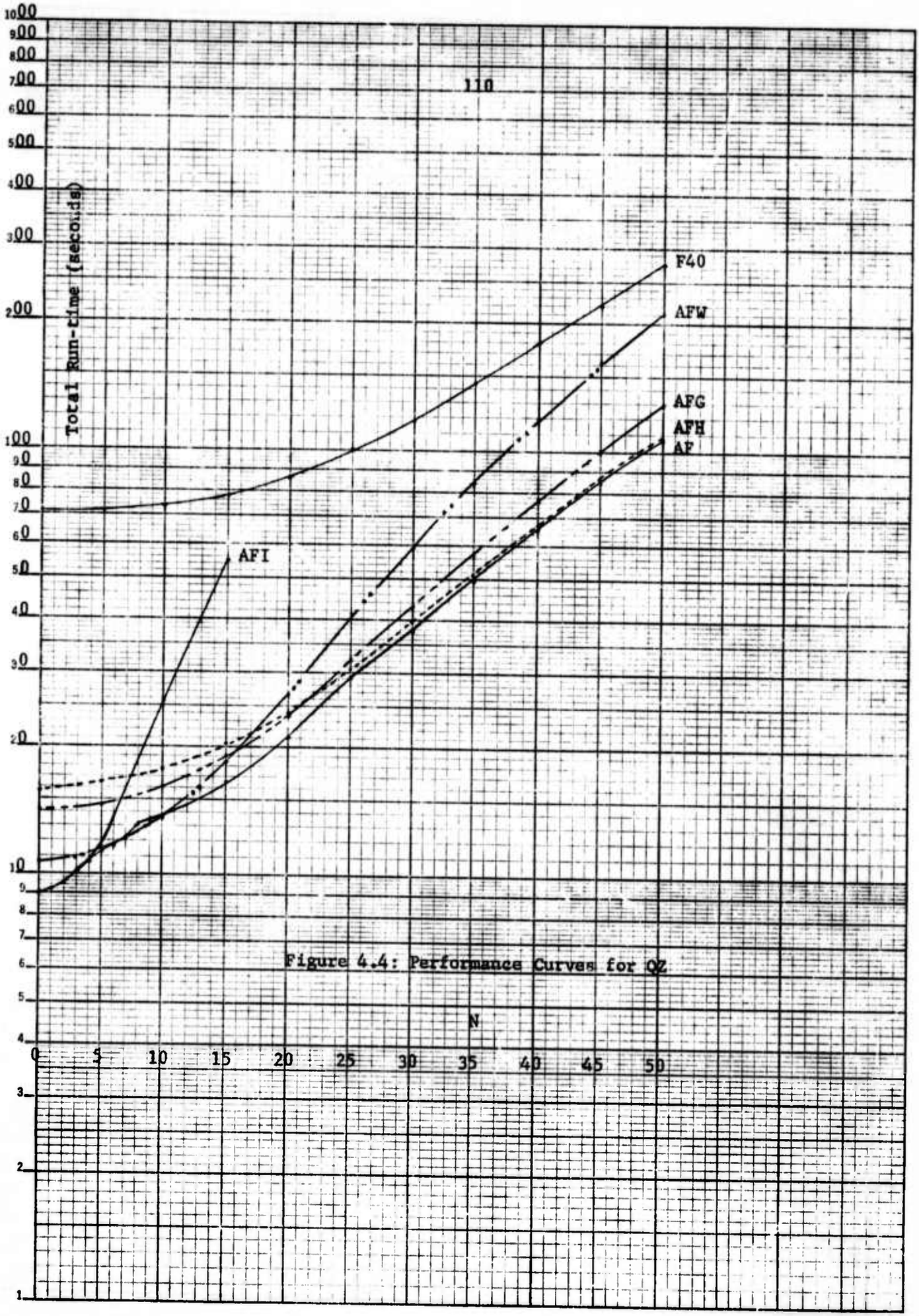


Figure 4.4: Performance Curves for QZ

Chapter V

Conclusions

This dissertation investigated the possibility of improving the cost effectiveness of code optimization. Whereas current approaches apply code optimization equally to the entire program at compile-time, our approach exploits dynamically the observed behavioral characteristics of programs, viz., that a small part (5%) of the code accounts for a large portion (50%) of the execution time. We studied, in general, the problems of performing code optimizations at run-time, i.e., dynamically determining which sections of code to optimize, how much optimization to apply, and when to apply that optimization. This resulted in the specification of a number of adaptive schema. The most promising scheme was incremental dynamic optimization which uses optimization counts to determine which section of code to optimize and when. The effect is gradual optimization of a program, i.e., one optimization is applied to one section of code at a time. The longer the program executes, the more optimized a section becomes. Using this scheme, a prototype system was built for an interesting subset of the FORTRAN language. Performance of this system, Adaptive FORTRAN (AF), was measured on a representative set of programs. In order to make unbiased comparisons with existing compiler systems, the adaptive system was transformed into various "normal" compiler systems that generate code analogous to that produced by WATFIV, FORTRAN-IV G and FORTRAN-IV H. The same set of

test programs were run under these transformed systems, and the performance measurements compared against those for AF.

The results were very encouraging. While AF did not outperform each of the other systems at all points in the run-time spectrum, it did perform better over the spectrum than did any other single compiler system. The major remaining problem lies in controlling the rate of optimization. AF's performance curves look worst for small-medium run-times, indicating too much optimization is being applied too soon. More research is needed to find a better means for controlling the optimization rate.

AF is the last of an evolutionary chain of experimental systems and there is every reason to believe it is possible to construct other variants which control optimization better and outperform all fixed-strategy systems everywhere in the run-time spectrum. The first line of attack should be to continue working with optimization counts. The method for estimating optimization counts presented in Section 2.5, viz., using the performance curves, E(q), for each optimization, should be explored. It would not be difficult to obtain such curves. Determining optimization counts heuristically has its limitations, for we found it hard to change an optimization count so only a portion of the performance curve is affected. Therefore, if any appreciable progress is to be made, a more theoretical basis for determining them must be developed. After this line of attack has been exhausted, other computationally feasible mechanisms and/or parameters for controlling the rate

of optimization should be explored.

In order to evaluate how good the incremental dynamic optimization scheme is, and determine exactly how much better we can expect to do, the absolute measure of performance should be obtained, for each test program, using the iterative dynamic optimization scheme. Using a large amount of computing effort, this performance curve can be obtained in the following manner. First, certain measurements must be made. For each optimizer, this consists of determing its performance curve, E(q), and its space requirements. For each segment of the program, measurements of its execution time and space requirements in all possible representations must be made. Using these measurements, optimal policies can be determined at run-time over the execution spectrum. But since it is not known when to determine such policies, they would have to be continuously determined, say after the execution of a basic block or segment, or a quantum of execution time. When the policy changes determines when to optimize. Using these results, the program would then be run for a given test point, policies changed at the appropriate time, and its execution time measured. The entire performance curve for the program can be obtained in this manner. The resulting curve does not contain the time required to determine the optimal policy; therefore it is the absolute best one can expect from any strategy.

We feel that we have demonstrated a worthwhile alternative to compiler design that should be considered seriously. The approach makes more sense,

from an implementation viewpoint, than building many special purpose compilers. The system can be built in an incremental fashion because of its modularity. Each step consists of programming and debugging an optimizer module and then adding it to the system. The final product is an adaptive compiler that does not require much more effort to build than a full optimizing compiler. AF was built in this manner: in 3 man months we had programmed the compiler and interpreter, and had programs running. Then each optimizer was programmed in 1/2 to 1 man month, debugged and added to the system. In less than a man year, the system was completed.

It is clear that such an implementation approach is open ended up to a point, for one can keep improving the efficiency of the generated code by adding more efficient optimizations until one exhausts optimizations. There are other well defined optimizations that work with the same internal form we produce; they should be added to the system and the performance measurements retaken, e.g., strength reduction, opening subroutines, and other machine dependent optimizations. There is one problem associated with adding more optimizations that became apparent as we constructed AF, viz., controlling the rate of optimization becomes harder. The means of control must be defined more sharply. This is the main reason why AF's optimization counts are determined as a function of the programs loop structure. As each optimizer was added, it became apparent that basic blocks and segments could no longer be treated uniformly with respect to the optimization counts.

Thus we see that additional research is needed to more clearly understand dynamic optimization and to refine the current approach. However, other areas are suggested on which further research should be conducted. It would be interesting to see if some hardware features can be developed to aid in controlling optimization. One beneficial feature would be for determining which section of code is being executed the most. There is one existing hardware feature we have not exploited for improving execution time, viz., micro-programming. There are two areas in the system that could utilize this feature. One is in interpreting the internal form produced by the compiler. Instead of writting a program to interpret it, micro-code could be developed for each operation. The other area is in the machine language generators. Instead of generating optimized code, specialized micro-code could be generated that performs the operations more efficiently.

Finally, the implications of our ideas should be studied with respect to conversational languages as indicated by Mitchell [Mit70]. He stated that a major problem in designing an interactive programming system is determining how to get efficiency and flexibility, two opposing constraints, to co-exist. His solution was to build an interpreter/compiler system. In such a system, a program is partially interpreted (to provide flexiblity for the user) or compiled (to provide efficient use of the computer) depending on its usage and constancy over some period of time. We see no conceptual problem in incorporating dynamic optimization into such a system in order to further improve efficiency. All that we would be doing is replacing the mechanism

that controls the compilation of code with a more refined one. Whenever changes are made to the program, the internal form used by the interpreter could be regenerated for those sections of the program affected, and their optimization state reset so they would be executed interpretively. As the program executes, these code sections would again be dynamically optimized.

In summary then, our test results indicate that the adaptive process is a worthwhile and promising technique. As our understanding of program behavior increases and our programming styles become more formalized, it may turn out to be one of the most sensible approaches for designing compiler systems.

Appendix A

The Compiled Code

To aid in syntax analysis, optimization and code generation, the compiler translates the source code into an internal form. A number of internal forms are possible: Polish notation, quadruples, triples, indirect triples or trees (cf. Gries [Gri71]). Optimizing compilers have been built using different internal forms, viz., FORTRAN IV H [Low69] uses quadruplets, FORTRAN II [All69] uses indirect triples, and BLISS [Bli71] uses trees. Which form to use is a matter of taste.

The adaptive FORTRAN compiler uses two internal forms. The compile-time internal form is Polish postfix which is used for syntax analysis and code generation. The run-time internal form is the generated code and consists of quadruples†, or quads for short. This form is an expanded version of the smaller and more consise source code in which language constructs (e.g., DOs, IFs, subscripts, tests) are expressed as basic operations.

A.1 Quadruples

There are a number of reasons why quads were selected as the run-time internal form. The main reason was that they were a convenient form that could be efficiently processed by the optimizers and executed

---

† Also known as three address code.

interpretively. Other reasons were:

1) A quad is self contained, i.e., it is not necessary to reference the result of another quad when processing its arguments.

2) Quads appear in the order in which they are to be executed.

3) Functions will know precisely where to return their results.

For a single binary operator, quads have the form:

$$(OP, ARG_1, ARG_2, ARG_3)$$

where ARG_1 and ARG_2 specify the operands, ARG_3 the result temporary, and OP the operation to be performed. Not all operations require three arguments; some require one (e.g., branches) while others two (e.g., conversions of type and unary operators). As a convention, unused positions of a quad are left blank.

A.2 Code Generated for each FORTRAN Construct

The adaptive FORTRAN compiler is one pass and generates relocatable interpretive code(i.e., quads) directly to core. If the program contains no errors, the relocatable code is loaded by a fast loader which maps relocatable addresses into absolute addresses and allocates data storage.

The generated code for some of the FORTRAN constructs is strictly quads (e.g., arithmetic operations); others are a combination of quads and machine language (e.g., calls to mathematical functions); while others are pure

machine language (e.g., I/O). Thus when the program is loaded, the instruction storage consists of quads with possible embedded machine language and pure machine language compiled out of sequence.

The descriptions of the generated code given below use the following programming conventions:

1) PDP 10 machine language is represented in MACRO-10 assembly language (cf. [PDP71a]).

2) A colon following a symbol indicates the symbol is a label.

3) The character '*' preceding a symbol indicates indirect addressing†.

4) A period following a symbol indicates it represents a FORTRAN UUO (cf. [PDP71b]), i.e., a call on the FORTRAN run-time support system.

5) A period represents the current address.

6) For arithmetic operations, the basic mnemonic has a single letter prefix to indicate the arithmetic mode:

   a) no prefix - integer OP
   b) F - floating point OP
   c) D - double precision OP
   d) C - complex OP
   e) L - logical OP
   f) S - string OP

   A complete list of the OP mnemonics is given in Table A.1 along with a brief description.

7) The meta-language variables used in the syntactic forms are the same as those used in the American Standard Fortran Report [ASF66]. Their meanings are generally obvious from context.

---

† MACRO-10 uses a '@' symbol instead, a convention we will not follow.

8) The subscripted letter T as an argument of a quad represents a temporary.

9) addr(v) represents the address of v.

10) Formatted data words are specified by the pseudo-ops DATA, DESC1, DESC2, DESC3 and TEXT. Their internal representations are given in Figure A.1.

A.2.1 Expressions

A. Aritmetic

a) Binary operator

FORM: e_1 <bop> e_2

CODE: (OP, e_1, e_2, T)

where <bop> ::= +|-|*|/|**

The OP mnemonics can be found in Table A.1.

b) Negation

FORM: -e

CODE: (NEG, e, , T)

B. Relational

FORM: e_1 <rop> e_2

CODE: (OP, e_1, e_2, T)

where <rop> ::= .LT.|.LE.|.EQ.|.NE.|.GE.|.GT.

The OP mnemonics can be found in Table A.1.

C. Logical

a) Binary operator

FORM: e_1 <lop> e_2

CODE: (OP, e_1, e_2, T)

where <lop> ::= .AND.|.OR.|.XOR.|.EQV.

The OP mnemonics can be found in Table A.1.

b) Unary .NOT.

FORM: .NOT. e

CODE: (NOT, e, , T)

A.2.2 Assignment Statement

FORM: $v_1 = v_2 = \ldots = v_n = e$

CODE: (REPL, e, , v_j) j=1,...,n

A.2.3 Control Statements

A. GO TO statements

a) Unconditional

FORM: GO TO k

CODE: (B, *k', ,)
k': DATA addr(k) (into data storage)

b) Assigned

FORM: GO TO v

CODE: (B, *v, ,)

c) ASSIGN statement

FORM: ASSIGN k TO v

where v is a simple variable.

CODE: (REPL, k, , v)

B. IF statement

a) Arithmetic

FORM: IF(e)k_1,k_2,k_3

where k_i is a statement label or assigned variable.

1) $k_1 \neq k_2 \neq k_3$

```
CODE:   (BLZ, e, *k1', )
        (BEZ, e, *k2', )
        (B, *k3', , )
   k1': DATA addr(k1)
   k2': DATA addr(k2)          (into data storage)
   k3': DATA addr(k3)
```

2) $k_1 = k_2$

```
CODE:   (BGZ, e, *k3', )
        (B, *k1', , )
```

3) $k_1 = k_3$

```
CODE:   (BEZ, e, *k2', )
        (B, *k1', , )
```

4) $k_2 = k_3$

```
CODE:   (BLZ, e, *k1', )
        (B, *k2', , )
```

b) Logical

```
FORM:   IF(e)S

CODE:   (BF, e, L, )
        {code for S}
     L: |
```

1) S is GO TO k

```
CODE:   (BT, e, *k', )
```

2) S is STOP

CODE: (STOPT, e, ,)

3) S is RETURN

CODE: (EXTFT/EXTST, e, ,)

C. Subprogram call

a) Subroutine

1) FORM: CALL s

CODE: (CALLS, *s', L1,)
L1: DATA 0 (out of sequence)
s': DATA addr(s) (into data storage)

2) FORM: CALL $s(a_1,a_2,...,a_n)$

CODE: (CALLS, *s', L1,)
L1: DATA n
DESC1 $TY_1,AR_1,L_1,addr(a_1)$
.
.
.
DESC1 $TY_n,AR_n,L_n,addr(a_n)$

where TY_j is the type of a_j (see Table A.2),
AR_j is the arithmetic of a_j (see Table A.3),
L_j is the class of a_j (see Table A.4).

b) Function

FORM: $f(a_1,a_2,...,a_n)$

CODE: (CALLF, *f', L1, T)
L1: DATA n
DESC1 $TY_1,AR_1,L_1,addr(a_1)$
.
.
.
DESC1 $TY_n,AR_n,L_n,addr(a_n)$

where T is the temporary storage location where the functional value is to be returned.

c) Basic external library function

FORM: xlf(a_1,...a_n)

CODE: (XCT, xlf , , T)
ARG <type code of a_1>,addr(a_1)
.
.
.
ARG <type code of a_n>,addr(a_n)

where ARG has the same format as I/O UUO's (see Sec. A.2.4). T is the temporary storage location where the functional value is to be returned.

D. RETURN statement

FORM: RETURN

a) Subroutine

CODE: (EXITS, , ,)

b) Function

CODE: (EXITF, f_v, ,)

where f_v is the address of the functional value.

E. DO statement

FORM: DO k v=e_1,e_2,e_3

where v is a simple variable,
e_j are arithmetic expressions which are converted to the type of v. e_3 may be omitted, in which case it is 1.

a) e_3 not a constant

CODE: (REPL, e_1, , v)
L1: {range of DO}
(ADD, v, e_3, v)
(SUB, v, e_2, T_1)
(NEGL, T_1, e_3, T_2)
(BLEZ, T_2, L1,)

b) e_3 a constant

1) $e_3 \geq 0$

```
CODE:   (REPL, e1, , v)
    L1: {range of DO}
        (ADD, v, e3, v)
        (BLE, v, e2, L1)
```

2) $e_3 < 0$

```
CODE:   (REPL, e1, , v)
    L1: {range of DO}
        (ADD, v, e3, v)
        (BGE, v, e2, L1)
```

F. CONTINUE statement

FORM: CONTINUE

CODE: none

G. END statement

FORM: END

CODE: (STOP, , ,) (only for the main program)

A.2.4 I/O Statements

I/O is performed by the PDP-10 FORTRAN I/O package. Since this section of code is fixed, it can not be optimized at run time. Hence I/O time is constant regardless of the optimizations made to the code. It would be wasteful (timewise) to have to transform interpretive I/O code to machine language. Therefore, the compiled code is identical to that produced by the PDP-10 FORTRAN compiler, F40. For a description of the FORTRAN UUO's IN., OUT., DATA., SLIST. and FIN., and ARG and type codes, see the PDP-10

FORTRAN handbook [PDP71b].

In what follows, the code is given in MACRO-10 format. Also, R0 and R1 represent machine registers 0 and 1 respectively.

A. Initialization

```
CODE:   MOVE    R1,<format pointer>
```

a) Input

```
FORM:   READ f,list
        READ f
        READ(u,f)list
        READ(u,f)
        READ(u,f,END=c)list
        READ(u,f,ERR=d)list
        READ(u,f,END=c,ERR=d)list

CODE:   IN.     R1,<unit number>
   or   MOVE    R0,<integer variable>
        HRRM    R0, .+1
        IN.     R1,0
        (if ERR or END specified)
        MOVE    R0,<label pointer>
        HRRM    R0,*<END/ERR>
```

where END/ERR are cells containing the address of END. and ERR., the cells used by the I/O package.

b) Output

```
FORM:   PRINT f,list
        PRINT f
        TYPE f,list
        TYPE f
        WRITE(u,f)list
        WRITE(u,f)

CODE:   OUT.    R1,<unit number>
   or   MOVE    R0,<integer variable>
        HRRM    R0, .+1
        OUT.    R1,0
```

B. Data transmission and I/O lists

FORM: $E_1,E_2,...,E_n$

where E_j can be a simple variable, subscripted variable, expression or array name, but not a DO-implied list.

a) Simple variable, constant, or expression (result)

1) non-parameter

```
CODE:   DATA.   <type code>,<variable/constant/result>
```

2) parameter

```
CODE:   DATA.   <type code>,*<parameter cell>
```

b) Array

1) non-adjustable dimensions

```
CODE:   SLIST.  <type code>,<base address of array>
        ARG     0,<number of elements>
```

2) adjustable dimensions

```
CODE:   MOVE    R0,addr(DVEC)+n+1
        HRRM    R0, .+2
        SLIST.  <type code>,*<parameter cell>
        ARG     0,0
```

where DVEC is the array's dope vector (see Sec. A.2.5).

c) Subscripted variable

```
CODE:   DATA.   <type code>,*<temp storage cell>
```

where <temp storage cell> contains the address of the array element.

C. Termination

```
CODE:   FIN.    0,0
```

Since the I/O code is in machine language, it cannot be mixed with the interpretive code. Therefore it is compiled out of sequence under a different relocation base. In order to execute it, it is made into a subroutine:

```
<I/O routine>:    BYTE    0
                  {I/O code}
                  JRST    2,*<I/O routine>
```

To execute the routine, the following quad is compiled:

(JSR, <I/O routine>, ,) .

The effect of the JSR is the execution of the machine language instruction:

JSR 0,<I/O routine> .

When the JSR quad is transformed to machine language, the JSR machine language instruction is generated.

D) FORMAT statement

FORM: k FORMAT(S_1,S_2,...,S_n)

CODE:

```
     k': DATA   addr(k)              (into data storage)
     k:  TEXT   '(S1,S2,...,Sn)'     (out of sequence)
```

A.2.5 Array Declarations

An array declarator may appear in a DIMENSION statement, type declaration or COMMON statement.

FORM: $v(d_1,d_2,...,d_n)$

where d_j are integers or simple integer variables,
n is the dimension of the array.

```
CODE:   (generated for arrays with adjustable dimensions)
        (PUSHJ, ADEC, , )
        DATA    n
        DATA    addr(DVEC)
        DESC2   R1,addr(d1)
          .
          .
          .
        DESC2   Rn,addr(dn)
```

where

a) ADEC is the run time array declaration routine which generates the dummy array's dope vector, DVEC. The dope vector has the form:

```
DVEC:   DATA    FUDGE
        DATA    D2
          .
          .
          .
        DATA    Dn
        DATA    SIZE        (number of elements)
```

b) R_j is the reference of d_j (see Table A.5).

c) If v is a dummy parameter, its value will be set by the run-time routine PSA and depends on the corresponding actual parameter. If the actual parameter is:

1) a subscripted variable, PSA stores the address of this element into v,

2) an array name, PSA stores the $BASE_v$ of the array into v. If the array name is not itself a parameter, its descriptor to PSA contains the $BASE_v$; if a parameter, the parameter cell contains the $BASE_v$.

A.2.6 Array References

References to array elements must contain the number of subscripts that corresponds to the number of dimensions declared for the array. Element $v(e_1,e_2,...,e_n)$ is at location:

$$BASE_v + (e_1*D_1+...+e_n*D_n) + FUDGE_v \quad (1)$$

where

a) $BASE_v$ is the address of the first element of the array v which has $e_1*...*e_n$ elements.

b) D_i is defined recursively as follows:

$$D_1 = 1$$
$$D_i = e_{i-1}*D_{i-1}$$

c) $FUDGE_v = -(D_1+...+D_n)$

A) Array with non-adjustable dimensions

In this case, all the information necessary to evaluate (1) at compile-time is stored in the dictionary along with the array's data descriptor.

1) Array not a dummy parameter

n=1 (ADD, e_1, F_v, T_1)

n>1

(MPY, e_2, D_2, T_1)
(ADD, T_1, e_1, T_2)
(MPY, e_3, D_3, T_3)
(ADD, T_3, T_2, T_4)
.
.
.
(MPY, e_n, D_n, T_{2n-3})
(ADD, T_{2n-3}, T_{2n-4}, T_{2n-2})
(ADD, T_{2n-2}, F_v, T_{2n-1})

where $F_v = BASE_v + FUDGE_v$

2) Array a dummy parameter

Replace the last instruction above with:

(ADD, T_{2n-2}/e_1, v, T_{2n-1})
(ADD, T_{2n-1}, $FUDGE_v$, T_{2n})

where v is the dummy parameter whose value is the BASE of the actual parameter,

$FUDGE_v$ is the FUDGE for the dummy array parameter calculated from its declaration at compile time.

B) Array with adjustable dimensions

n=1

(ADD, v, DVEC, T_1)
(ADD, T_1, e_1, T_2)

n>1

(ADD, v, DVEC, T_1)
(ADD, T_1, e_1, T_2)
(MPY, e_2, DVEC+1, T_3)
(ADD, T_3, T_2, T_4)
.
.
.
(MPY, e_n, DVEC+n-1, T_{2n-1})
(ADD, T_{2n-1}, T_{2n-2}, T_{2n})

where v is the dummy parameter whose value is the BASE of the actual parameter.

A.2.7 Subprograms

A. FUNCTION Subprograms

FORM: t FUNCTION $f(a_1,a_2,...,a_n)$

where t is optional and can be INTEGER, REAL or LOGICAL,
a_j is a dummy parameter.

Functions must have at least one dummy parameter. A RETURN statement must be supplied. The name of the function is treated as a scalar variable for storing the value of the function. Storage for the functional value is allocated as for normal scalars.

Functions are referenced within expressions and return a value. The code generated for a function reference is given in Section A.2.3.

B. SUBROUTINE Subprograms

FORM: SUBROUTINE s

or SUBROUTINE $s(a_1,a_2,...,a_n)$

where a_j is a dummy parameter.

C. Code generated for a subprogram definition

CODE:
```
(PUSHJ, PSA, , )
DATA    n
DESC3   TY1,AR1,psi1
  .
  .
  .
DESC3   TYn,ARn,psin
```

where PSA is the run time parameter assignment routine,
TY_j is the type of a_j (see Table A.2),
AR_j is the arithmetic of a_j (see Table A.3),
psi_j is the parameter storage index for a_j.

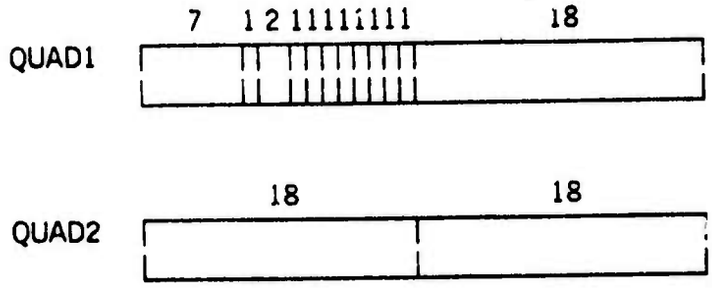
PSA matches the actual parameters with the formal parameters. Since all parameters are call by address, no conversion of type is possible. Therefore arithmetics must match. Using {TY,AR,L} in the subprogram reference and {TY,AR} in the subprogram definition, PSA calculates the address of the actual parameter and inserts it into the corresponding psi. Thus, references to the actual parameter is indirect through its psi.

A.3 Internal Representation of Quads

The internal representation for quads on the PDP-10 requires two 36-bit words:

QUAD1 OP,SR,BTY,C_1,C_2,T_1,T_2,T_3,I_1,I_2,I_3,addr(ARG_1)
QUAD2 addr(ARG_2),addr(ARG_3)

having the format:



The 11-bit control field after OP is a set of tag bits which represent information about the data found in the associated ARG or the address type of the associated ARG. Tag bits I_1-I_3 are set by the compiler; tag bits C_1-C_2, T_1-T_3, and SR are set by the loader; and tag bit BTY is set by the machine language generators.

The function of each tag bit is:

1) Indirect addressing indicators I_1-I_3

If I_j is 0, ARG_j is the address of the operand; if I_j is 1, ARG_j contains the address of the operand (indirect result or parameter).

2) Temporary address indicators T_1-T_3

T_j is 1 if ARG_j is the address of a temporary; otherwise 0. These indicators exist for efficiency purposes. Temporarys are the most heavily processed entities, and even though it is possible at run time to determine if an address represents a temporary, to do so would increase the processing overhead needlessly.

3) Constant address indicators C_1-C_2

C_j is 1 if ARG_j is the address of a constant. Again these indicators exist for efficiency purposes. They aid the machine language generators in determining if it is possible to use an "immediate" instruction.

4) Branch type indicator BTY

Set whenever the branch is translated or retranslated to machine language to insure the proper code is generated (see Section 3.4). This tag bit is applicable only to branch instructions. Basically it is used to distinguish whether the branch is to a basic block that is external or internal to the segment containing it. It must be updated whenever new segments are formed or optimizations applied.

5) Store result temporary indicator SR

If SR is 1, the machine language generator compiles a store instruction to force the storing of the temporary's associated register into the temporary. This is necessary when a temporary is referenced in machine language generated by the compiler. This machine language is never altered, and consequently when the quad is translated to machine language, its result must be stored in the result temporary.

Table A.1 The List of Quad OP codes

Octal	Mnemonic	Description	
000	NOP	No operation	
001	ADD	Integer add	$ARG_3 \leftarrow ARG_1 + ARG_2$
002	FADD	Floating add	
003	DADD	Double precision add	
004	CADD	Complex add	
005	SUB	Integer subtract	$ARG_3 \leftarrow ARG_1 - ARG_2$
006	FSUB	Floating subtract	
007	DSUB	D.P. subtract	
010	CSUB	Complex subtract	
011	MPY	Integer multiply	$ARG_3 \leftarrow ARG_1 * ARG_2$
012	FMPY	Floating multiply	
013	DMPY	D.P. multiply	
014	CMPY	Complex multiply	
015	DIV	Integer divide	$ARG_3 \leftarrow ARG_1 / ARG_2$
016	FDIV	Floating divide	
017	DDIV	D.P. divide	
020	CDIV	Complex divide	
021	FXFX	Integer to integer power	$ARG_3 \leftarrow ARG_1 ** ARG_2$
022	FLFL	Floating to floating power	
023	DPDP	D.P. to D.P. power	
024	CXCX	Complex to complex power	
025	FLFX	Floating to integer power	
026	DPFX	D.P. to integer power	
027	CXFX	Complex to integer power	
030	NEG	Integer negate	$ARG_3 \leftarrow -ARG_1$
031	FNEG	Floating negate	
032	DNEG	D.P. negate	
033	CNEG	Complex negate	
034	LREPL	Logical replacement	$ARG_3 \leftarrow ARG_1$
035	SREPL	String replacement	
036	REPL	Integer replacement	
037	FREPL	Floating replacement	
040	DREPL	D.P. replacement	
041	CREPL	Complex replacement	
042	AND	Logical and	$ARG_3 \leftarrow ARG_1 \wedge ARG_2$
043	NOT	Logical not	$ARG_3 \leftarrow \neg ARG_1$
044	OR	Logical or	$ARG_3 \leftarrow ARG_1 \vee ARG_2$
045	XOR	Logical exclusive or	$ARG_3 \leftarrow ARG_1$ xor ARG_2
046	EQV	Logical equivalence	$ARG_3 \leftarrow ARG_1 \equiv ARG_2$
047	SEQ	String =	$ARG_3 \leftarrow (ARG_1 = ARG_2)$
050	EQ	Integer =	
051	FEQ	Floating =	
052	DEQ	D.P. =	

Table A.1 (cont.)

Octal	Mnemonic	Description	
053	CEQ	Complex =	
054	SNE	String ≠	$ARG_3 \leftarrow (ARG_1 \neq ARG_2)$
055	NE	Integer ≠	
056	FNE	Floating ≠	
057	DNE	D.P. ≠	
060	CNE	Complex ≠	
061	SGT	String >	$ARG_3 \leftarrow (ARG_1 > ARG_2)$
062	GT	Integer >	
063	FGT	Floating >	
064	DGT	D.P. >	
065	SGE	String ≥	$ARG_3 \leftarrow (ARG_1 \geq ARG_2)$
066	GE	Integer ≥	
067	FGE	Floating ≥	
070	DGE	D.P. ≥	
071	SLT	String <	$ARG_3 \leftarrow (ARG_1 < ARG_2)$
072	LT	Integer <	
073	FLT	Floating <	
074	DLT	D.P. <	
075	SLE	String ≤	$ARG_3 \leftarrow (ARG_1 \leq ARG_2)$
076	LE	Integer ≤	
077	FLE	Floating ≤	
100	DLE	D.P. ≤	
101	MOD	Integer **mod**	$ARG_3 \leftarrow ARG_1$ **mod** ARG_2
102	AMOD	Floating **mod**	
103	ISIGN	Integer **sign**	$ARG_3 \leftarrow$ **sgn**$(ARG_1) * \lvert ARG_2 \rvert$
104	SIGN	Floating **sign**	
105	DSIGN	D.P. **sign**	
106	IABS	Integer **abs**	$ARG_3 \leftarrow \lvert ARG_1 \rvert$
107	ABS	Floating **abs**	
110	DABS	D.P. **abs**	
111	CABS	Complex **abs**	
112	INT	Real to integer truncation	$ARG_3 \leftarrow$ **sgn**$ARG_1 *$ **entier** $\lvert ARG_1 \rvert$
113	AINT	Real to real truncation	
114	IDINT	D.P. to integer truncation	
115	IFIX	Real to integer conversion	$ARG_3 \leftarrow$ **entier** ARG_1
116	FLOAT	Integer to real conversion	
117	CVSI	String to integer conversion	
120	CVSR	String to real conversion	
121	CVSD	String to D.P. conversion	
122	CVSC	String to complex conversion	
123	B	Branch to ARG_1	
124	BGZ	Branch to ARG_2 if $ARG_1 > 0$	
125	BF	Branch to ARG_2 if ARG_1=**false**	

Table A.1 (cont.)

Octal	Mnemonic	Description
126	BLZ	Branch to ARG_2 if $ARG_1<0$
127	BEZ	Branch to ARG_2 if $ARG_1=0$
130	BLEZ	Branch to ARG_2 if $ARG_1\leq 0$
131	STOP	Stop execution
132	NEGL	Integer conditional negate $ARG_3 \leftarrow$ if $ARG_2<0$ then $-ARG_1$ else ARG_1
133	FNEGL	Floating conditional negate
134	DNEGL	D.P. conditional negate
135	EXITS	Return from subroutine
136	BLE	Branch to ARG_3 if $ARG_1\leq ARG_2$
137	BGE	Branch to ARG_3 if $ARG_1\geq ARG_2$
140	CALLF	Call the function at ARG_1. ARG_3 is the temporary for the functional value. ARG_2 is the address of the formal parameter descriptor list.
141	EXITF	Return from function. ARG_1 contains the functional value.
142	JSR	Simulate PDP-10 JSR instruction. The routine is at ARG_1 (used to call I/O subroutines).
143	XCT	Simulate PDP-10 XCT instruction. Instruction to be executed is at ARG_1 (used to call external library functions). ARG_3 is the functional result.
144	PUSHJ	Simulate PDP-10 PUSHJ instruction. The stack used is the BLISS run time stack [Bli71]. The routine to be called is at ARG_1 (used to call the run-time support routines ADEC and PSA).
145	JUMP	Branch to ARG_1 (marks end of a basic block)
146	CALLS	Call subroutine at ARG_1. ARG_2 is the address of the formal parameter descriptor list.
147	STOPT	Stop execution if ARG_1=true
150	EXTST	Return from subroutine if ARG_1=true
151	EXTFT	Return from function if ARG_1=true. ARG_2 contains the functional value.
152	BT	Branch to ARG_2 if ARG_1=true

Table A.2 Operand Type (TY)

Octal	Type
00	simple variable
10	array with non-adjustable dimensions
11	array with adjustable dimensions
20	function subprogram
21	subroutine subprogram
22	library subprogram
23	external subprogram

Table A.3 Operand Arithmetic (AR)

Octal	Arithmetic
0	universal
1	logical
2	string
3	integer
4	real
5	double precision
6	complex

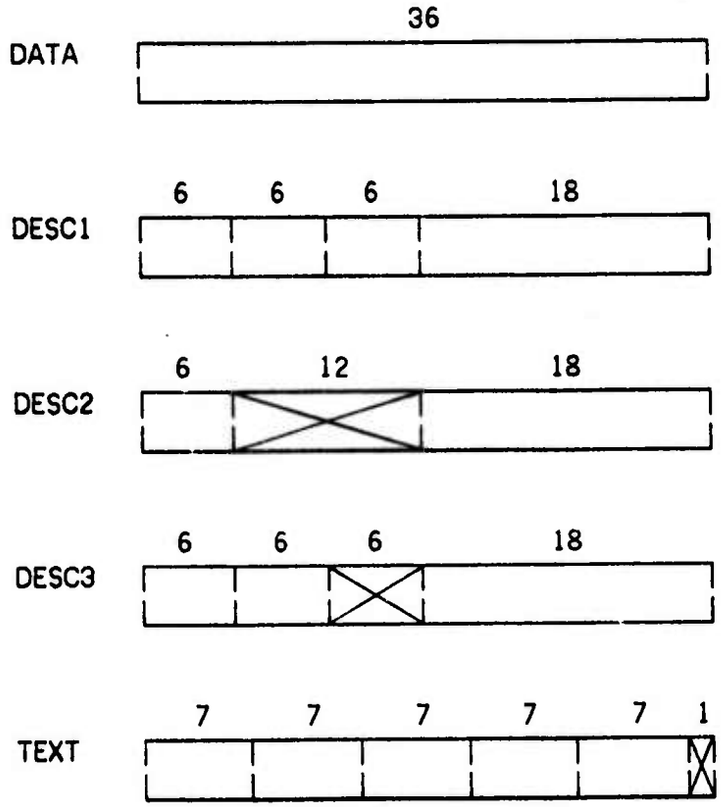
Table A.4 Operand Class (L)

Octal	Class
1	identifier
2	constant
3	result
4	indirect result
5	parameter

Table A.5 Operand Reference (R)

Octal	Reference
0	normal variable
1	COMMON variable
2	parameter

Figure A.1 Internal Representations of Formatted Data Words



Appendix B

Source Listings of the Test Programs and a Detailed Example

This appendix contains the source listings of all the test programs used for validation of the system, along with the complete system output for a matrix multiplication program. This detailed example is the same as that used by Allen [All69].

B.1 A Detailed Example: Matrix Multiplication

A) The Source Listing

```
 1.          INTEGER X(50,50), Y(50,50), Z(50,50)
 2.    C     INITIALIZE X AND Y
 3.          DO 10 I=1,50
 4.          DO 10 J=1,50
 5.          X(I,J)=I+J
 6.          Y(I,J)=MOD(I,J)
 7.    10    CONTINUE
 8.          DO 3 I=1,50
 9.          DO 3 J=1,50
10.          Z(I,J)=0
11.          DO 3 K=1,50
12.    3     Z(I,J)=Z(I,J)+X(I,K)*Y(K,J)
13.          TYPE 20,X,Y,Z
14.    20    FORMAT(5(5I5/)/)
15.          STOP
16.          END
```

B) Listing of Source with Interpretive Code

```
 AFORTRAN VERSION: 110172        12/4/72 13:37.48          P3F4          PAGE 1-1

        03 00 00 00 000000   777777777777                          DATA    TRUE
        03 00 00 00 000001   000000000000                          DATA    FALSE
        *********BLOCK 1
        63 01 00 00 000001   000000000000                          DATA    000000000000
00100     1.          INTEGER X(50,50), Y(50,50), Z(50,50)
00200     2.    C     INITIALIZE X AND Y
00300     3.          DO 10 I=1,50
        03 00 00 00 000003   000000000001                          DATA    000000000001
00400     4.          DO 10 J=1,50
        03 00 00 00 000004   000000000062                          DATA    000000000062
        01 03 00 05 000000   140000000003 000000 000000            (REPL   ,00000000001 ,0          ,I          )
        01 00 00 00 000002   624000000002 000000 000000            (JUMP   ,000002      ,0          ,0          )
        *********BLOCK 2
        63 01 00 00 000002   000000000004                          DATA    000000000004
00500     5.          X(I,J)=I+J
        01 03 00 05 000004   176000000003 000000 000001            (REPL   ,00000000001 ,0          ,J          )
        01 00 00 00 000006   624000000003 000000 000000            (JUMP   ,000003      ,0          ,0          )
        *********BLOCK 3
        63 01 00 00 000003   000000000010                          DATA    000000000010
        03 05 00 00 000005   777777777717                          DATA    -000000000061
        01 05 03 04 000010   044000000001 000004 000000            (MPY    ,J           ,00000000050 ,T0$        )
        01 04 05 04 000012   004000000000 000000 000001            (ADD    ,T0$         ,I           ,T1$        )
        01 04 03 04 000014   004000000001 000003 000002            (ADD    ,T1$         ,-00000000043 ,T2$       )
00600     6.          Y(I,J)=M00(I,J)
        01 05 05 04 000016   004000000000 000001 000003            (ADD    ,I           ,J           ,T3$        )
        01 04 00 04 000020   170001000003 000000 000002            (REPL   ,T3$         ,0           ,*T2$       )
        03 05 00 00 000006   000000004623                          DATA    000000004623
        01 05 03 04 000022   044000000001 000004 000004            (MPY    ,J           ,00000000050 ,T4$        )
        01 04 05 04 000024   004000000004 000000 000005            (ADD    ,T4$         ,I           ,T5$        )
        01 04 03 04 000026   004000000005 000006 000006            (ADD    ,T5$         ,00000002451 ,T6$        )
00700     7.    10    CONTINUE
        01 05 05 04 000030   404000000000 000001 000007            (MOD    ,I           ,J           ,T7$        )
        01 04 00 04 000032   170001000007 000000 000006            (REPL   ,T7$         ,0           ,*T6$       )
00800     8.          DO 3 I=1,50
        01 00 00 00 000034   624000000004 000000 000000            (JUMP   ,000004      ,0          ,0          )
        *********BLOCK 4
        63 01 00 00 000004   000000000036                          DATA    000000000036
        03 00 00 00 000002   000000000004                  000010  DATA    000000000004
        01 05 03 05 000036   004000000001 000003 000001            (ADD    ,J           ,00000000001 ,J          )
        01 05 03 00 000040   570000000001 000004 000003            (BLE    ,J           ,00000000050 ,000003     )
        01 00 00 00 000042   624000000005 000000 000000            (JUMP   ,000005      ,0          ,0          )
        *********BLOCK 5
        63 01 00 00 000005   000000000044                          DATA    000000000044
        01 05 03 05 000044   004000000000 000003 000000            (ADD    ,I           ,00000000001 ,I          )
        01 05 03 00 000046   570000000000 000004 000002            (BLE    ,I           ,00000000050 ,000002     )
        01 00 00 00 000050   624000000006 000000 000000            (JUMP   ,000006      ,0          ,0          )
        *********BLOCK 6
        63 01 00 00 000006   000000000052                          DATA    000000000052
00900     9.          DO 3 J=1,50
        01 03 00 05 000052   170000000003 000000 000000            (REPL   ,00000000001 ,0          ,I          )
        01 00 00 00 000054   624000000007 000000 000000            (JUMP   ,000007      ,0          ,0          )
        *********BLOCK 7
        63 01 00 00 000007   000000000056                          DATA    000000000056
01000    10.          Z(I,J)=0
        01 03 00 05 000056   170000000003 000000 000001            (REPL   ,00000000001 ,0          ,J          )
        01 00 00 00 000060   624000000010 000000 000000            (JUMP   ,000010      ,0          ,0          )
        *********BLOCK 8
        63 01 00 00 000010   000000000062                          DATA    000000000062
        03 05 00 00 000010   000000011527                          DATA    000000011527
        01 05 03 04 000062   044000000001 000004 000000            (MPY    ,J           ,00000000050 ,T8$        )
        01 04 05 04 000064   004000000000 000000 000001            (ADD    ,T8$         ,I           ,T9$        )
        01 04 03 04 000066   004000000001 000010 000002            (ADD    ,T9$         ,00000004951 ,T10$       )
01100    11.          DO 3 K=1,50
        03 00 00 00 000011   000000000000                          DATA    000000000000
        01 03 00 04 000070   170001000011 000000 000002            (REPL   ,00000000000 ,0          ,*T10$      )
01200    12.    3     Z(I,J)=Z(I,J)+X(I,K)*Y(K,J)
        01 03 00 05 000072   176000000003 000000 016516            (REPL   ,00000000001 ,0          ,K          )
        01 00 00 00 000074   624000000011 000000 000000            (JUMP   ,000011      ,0          ,0          )
```

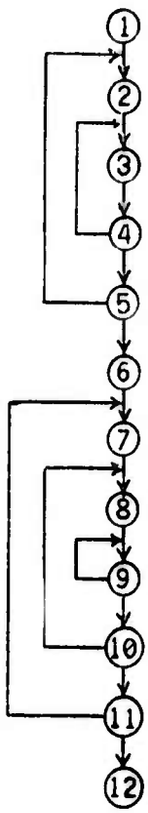
```
AFORTRAN VERSION: 110172        12/4/72 137.47          P3F4          PAGE 1-2

          **********BLOCK 9
          63 01 00 00 000011   000000000076                              DATA    000000000076
          03 00 00 00 000007   000000000011                    00003     DATA    000000000011
          01 05 03 04 000076   044000000001 000004 000000                (MPY   , J          , 00000000050 , T11$      )
          01 04 05 04 000100   004000000000 000000 000000                (ADD   , T11$       , I           , T12$      )
          01 04 03 04 000102   004000000001 000010 000000                (ADD   , T12$       , 00000004951 , T13$      )
          01 05 03 04 000104   044000000001 000004 000003                (MPY   , J          , 00000000050 , T14$      )
          01 04 05 04 000106   004000000003 000000 000004                (ADD   , T14$       , I           , T15$      )
          01 04 03 04 000110   004000000004 000010 000005                (ADD   , T15$       , 00000004951 , T16$      )
          01 05 03 04 000112   044000016516 000004 000006                (MPY   , K          , 00000000050 , T17$      )
          01 04 05 04 000114   004000000006 000000 000007                (ADD   , T17$       , I           , T18$      )
          01 04 03 04 000116   004000000007 000005 000010                (ADD   , T18$       , 00000000049 , T19$      )
01300       13.            TYPE 20,X,Y,Z
          01 05 03 04 000120   044000000001 000004 000011                (MPY   , J          , 00000000050 , T20$      )
          01 04 05 04 000122   004000000011 016516 000012                (ADD   , T20$       , K           , T21$      )
          01 04 03 04 000124   004000000012 000006 000013                (ADD   , T21$       , 00000002451 , T22$      )
          01 04 04 04 000126   044006000010 000013 000014                (MPY   , #T19$      , #T22$       , T23$      )
          01 04 04 04 000130   004004000005 000014 000015                (ADD   , #T16$      , T23$        , T24$      )
          01 04 00 04 000132   170001000015 000000 000002                (REPL  , T24$       , 0           , #T13$     )
          01 05 03 05 000134   004000016516 000003 016516                (ADD   , K          , 00000000001 , K         )
          01 05 03 00 000136   570000016516 000004 000011                (BLE   , K          , 00000000050 , 000011    )
          01 00 00 00 000140   624000000012 000000 000000                (JUMP  , 000012     , 0           , 0         )
          **********BLOCK 10
          63 01 00 00 000012   000000000142                              DATA    000000000142
          01 04 03 05 000142   004000000001 000003 000001                (ADD   , J          , 00000000001 , J         )
          01 05 03 00 000144   570000000001 000004 000010                (BLE   , J          , 00000000050 , 000010    )
          01 00 00 00 000146   624000000013 000003 000000                (JUMP  , 000013     , 0           , 0         )
          **********BLOCK 11
          63 01 00 00 000013   000000000150                              DATA    000000000150
          01 05 03 05 000150   004000000000 000003 000000                (ADD   , I          , 00000000001 , I         )
          01 05 03 00 000152   570000000000 000004 000007                (BLE   , I          , 00000000050 , 000007    )
          01 00 00 00 000154   624000000014 000000 000000                (JUMP  , 000014     , 0           , 0         )
          **********BLOCK 12
          63 01 00 00 000014   000000000156                              DATA    000000000156
          02 00 00 00 000000   000000000000                              DATA    000000000000
          02 03 00 00 000001   200040000012                              MOVE    01,000020$
          02 00 00 00 000002   017040777777                              OUT.    01,777777
01400       14.   20    FORMAT(5(5I5,/),/)
          02 05 00 00 000003   025000000002                              SLIST.  00,X
          02 00 00 00 000004   320000004704                              ARG     00,004704
          02 05 00 00 000005   025000004706                              SLIST.  00,Y
          02 00 00 00 000006   320000004704                              ARG     00,004704
          02 05 00 00 000007   025000011612                              SLIST.  00,Z
          02 00 00 00 000010   320000001704                              ARG     00,004704
          02 00 00 00 000011   021000000000                              FIN.    00,000000
          02 02 00 00 000012   254120000                                 JRST    02,#000000
          01 02 00 00 000156   610000                  00 000000         (JSR   , 0          , 0           , 0         )
          03 02 00 00 000012   000000                                00020 DATA    000000000013
          02 00 00 00 000013   241525092                                 TEXT    (5(5I
          02 00 00 00 000014   325365127522                              TEXT    5/),/)
01500       15.            STOP
01600       16.            END
          01 00 00 00 000160   544000000000 000000 000000                (STOP  , 0          , 0           , 0         )
          01 00 00 00 000162   624000000015 000000 000000                (JUMP  , 000015     , 0           , 0         )
          **********BLOCK 13
          63 01 00 00 000015   000000000164                              DATA    000000000164
          01 00 00 00 000164   544000000000 000000 000000                (STOP  , 0          , 0           , 0         )
          01 00 00 00 000166   624000000000 000000 000000                (JUMP  , 0          , 0           , 0         )
          01 00 00 00 000170   000000000001                              START   000000000001
```

C) Listing of Immediate Predecessors

BASIC BLOCK	IMMEDIATE PREDECESSOR(S)
1	NONE
2	1 5
3	2 4
4	3
5	4
6	5
7	6 11
8	7 10
9	8 9
10	9
11	10
12	11
13	NONE

D) The Directed Graph



E) Listing of Code Optimizations

The program was constructed so that the entire main loop would become totally optimized. The main loop consists of statements 8 thru 12, or basic blocks 6 thru 11. The partial listing of the optimized code given below is only for these blocks.

The optimization of the program can be summarized as follows:

1) Fusion of segment 9 (basic block 9)

Since the main loop consists of three nested DO loops, the corresponding segments will be optimized in the order they are embedded, starting with the innermost one. Thus, segment 9 is optimized first, and since it is the innermost segment, it is fused to "dumb" code after being executed once interpretively. Since the segment is not yet totally optimized, the conditional BLE branch to itself is processed by the segment driver so the segment's optimization count will be decremented.

2) Code Motion on segment 9

Segment 9 is now totally optimized. First CSE is performed on each of its basic blocks (one in this case). Four redundant subexpressions are removed from basic block 9: the 4th, 5th, 6th and 10th quads. The first three represent the second subscript calculations for Z(I,J), while the fourth involves the common subscript calculation for J. Also, the 15th quad is combined with the 14th, eliminating the intermediate temporary. Then code motion results in three calculations involving the segment invariants I and J being moved to the front of the segment: the 1st, 2nd, and 3rd quads. Unique temporaries are assigned to the results of these invariant quads, and they replace the original temporaries. Thus the result of the 1st quad is used both in the 2nd quad and the 11th quad, and the 3rd quads's result is used in the 14th quad. Finally, the

resulting quads are compiled to "fair" code.

Notice that the conditional BLE branch is now direct and to the alternate entry point of the segment, i.e., to the point after the invariant code.

3) Translating basic blocks 8 and 10

The remaining basic blocks of the yet unformed segment 8 are now translated to "dumb" machine language.

4) Fusion of segment 8 (basic blocks 8-10)

Next, segment 8 is formed. The machine language is non-homogenous with respect to the degree of optimization performed on its basic blocks: the embedded segment 9 is already totally optimized, while the rest of its blocks have been translated to "dumb" code. Since the machine language for all the segment's basic blocks exists, it is only necessary to combine the machine language for each block, at the same time retranslating the branches. Notice that no code is generated for the intra-segment JUMP's and the direct branch of segment 9 now reflects where the new alternate entry point is located.

5) Code Motion on segment 8

Finally, segment 8 is totally optimized. This means performing CSE on basic blocks 8 and 10, then code motion on the entire segment. These optimizations have no effect. When forming the machine language segment, basic blocks 8 and 10 are compiled to "fair" code, but since the machine language for segment 9 already exists, it need only be moved. The branches of each basic block are again retranslarted resulting in the unconditional BLE branch of block 10 being made direct.

6) Optimization of segment 7 (basic blocks 7-11)

The optimization for this segment proceeds as for segment 8 in a straight forward manner.

```
FUSED BLOCKS   9 THRU   9

TRANSLATING BLOCK   9
       032226    (MPY   , 033661 , 032341 , 032530)
                 053265    MOVEI   04, 000062
                 053266    IMUL    04, 033661
       032230    (ADD   , 032530 , 033660 , 032531)
                 053267    ADD     04, 033660
       032232    (ADD   , 032531 , 032345 , 032532)
                 053270    ADDI    04, 045407
       032234    (MPY   , 033661 , 032341 , 032533)
                 053271    MOVEI   05, 000062
                 053272    IMUL    05, 033661
       032236    (ADD   , 032533 , 033660 , 032534)
                 053273    ADD     05, 033660
       032240    (ADD   , 032534 , 032345 , 032535)
                 053274    ADDI    05, 045407
       032242    (MPY   , 052376 , 032341 , 032536)
                 053275    MOVEI   06, 000062
                 053276    IMUL    06, 052376
       032244    (ADD   , 032536 , 033660 , 032537)
                 053277    ADD     06, 033660
       032246    (ADD   , 032537 , 032342 , 032540)
                 053300    ADDI    06, 033577
       032250    (MPY   , 033661 , 032341 , 032541)
                 053301    MOVEI   07, 000062
                 053302    IMUL    07, 033661
       032252    (ADD   , 032541 , 052376 , 032542)
                 053303    ADD     07, 052376
       032254    (ADD   , 032542 , 032343 , 032543)
                 053304    ADDI    07, 040503
       032256    (MPY   ,@032540 ,@032543 , 032544)
                 053305    MOVE    06, 000000(06)
                 053306    IMUL    06,@000007
       032260    (ADD   ,@032535 , 032544 , 032545)
                 053307    MOVE    05, 000000(05)
                 053310    ADD     05, 000005
       032262    (REPL  , 032545 , 000000 ,@032532)
                 053311    MOVEM   05,@000004
       032264    (ADD   , 052376 , 032340 , 052376)
                 053312    MOVEI   07, 000001
                 053313    ADD     07, 052376
                 053314    MOVEM   07, 052376
       032266    (BLE   , 052376 , 032341 , 000011)
                 053315    MOVEI   15, 000011
                 053316    CAMG    07, 032341
                 053317    POPJ    17, 000000
       032220    (JUMP  , 000012 , 000000 , 000000)
                 053320    MOVEI   15, 000012
                 053321    POPJ    17, 000000
   QUADS TRANSLATED =  18

CODE MOTION ON SEGMENT   9

CSE ON BLOCK   9
   QUADS ELIMINATED =   4
   QUADS REMOVED =   3

COMPILING BLOCK   9
       053322    (MPY   , 033661 , 032341 , 032546)
                 053332    MOVE    04, 033661
                 053333    IMULI   04, 000062
                 053334    MOVEM   04, 032546
       053324    (ADD   , 032546 , 033660 , 032547)
                 053335    MOVE    04, 032546
                 053336    ADD     04, 033660
                 053337    MOVEM   04, 032547
       053326    (ADD   , 032542 , 032345 , 032550)
                 053340    ADDI    04, 045407
                 053341    MOVEM   04, 032550
       053330    (JUMP  , 000011 , 032226 , 032266)
       032226    (NOP   , 000000 , 000000 , 000000)
       032230    (NOP   , 000000 , 000000 , 000000)
       032232    (NOP   , 000000 , 000000 , 000000)
       032234    (NOP   , 000000 , 000000 , 000000)
       032236    (NOP   , 000000 , 000000 , 000000)
       032240    (NOP   , 000000 , 000000 , 000000)
       032242    (MPY   , 052376 , 032341 , 032536)
                 053342    MOVE    04, 052376
                 053343    IMULI   04, 000062
       032244    (ADD   , 032536 , 033660 , 032537)
                 053344    ADD     04, 033660
       032246    (ADD   , 032537 , 032342 , 032540)
       032250    (NOP   , 000000 , 000000 , 000000)
       032252    (ADD   , 032546 , 052376 , 032542)
                 053345    MOVE    05, 032546
                 053346    ADD     05, 052376
       032254    (ADD   , 032542 , 032343 , 032543)
       032256    (MPY   ,@032540 ,@032543 , 032544)
                 053347    MOVE    04, 033577(04)
                 053350    IMUL    04, 040503(05)
       032260    (ADD   ,@032550 , 032544 ,@032550)
                 053351    ADDB    04,@032550
       032262    (NOP   , 000000 , 000000 , 000000)
       032264    (ADD   , 052376 , 032340 , 052376)
                 053352    AOS     05, 052376
       032266    (BLE   , 052376 , 032341 , 000011)
                 053353    CAMG    05, 032341
                 053354    JRST    00, 053342
       032270    (JUMP  , 000012 , 000000 , 000000)
                 053355    MOVEI   15, 000012
                 053356    POPJ    17, 000000
   QUADS COMPILED =  14

TRANSLATING BLOCK   8
       032212    (MPY   , 033661 , 032341 , 032530)
                 053357    MOVEI   04, 000062
                 053360    IMUL    04, 033661
       032214    (ADD   , 032530 , 033660 , 032531)
                 053361    ADD     04, 033660
       032216    (ADD   , 032531 , 032345 , 032532)
                 053362    ADDI    04, 045407
       032220    (REPL  , 032346 , 000000 ,@032532)
                 053363    MOVEI   05, 000000
                 053364    MOVEM   05,@000004
       032222    (REPL  , 032340 , 000000 , 052376)
                 053365    MOVEI   05, 000001
                 053366    MOVEM   05, 052376
       032224    (JUMP  , 000011 , 000000 , 000000)
                 053367    MOVEI   15, 000011
                 053370    POPJ    17, 000000
   QUADS TRANSLATED =   6

TRANSLATING BLOCK   10
       032272    (ADD   , 033661 , 032340 , 033661)
                 053371    MOVEI   04, 000001
                 053372    ADD     04, 033661
                 053373    MOVEM   04, 033661
       032274    (BLE   , 033661 , 032341 , 000010)
                 053374    MOVEI   15, 000010
                 053375    CAMG    04, 032341
                 053376    POPJ    17, 000000
       032276    (JUMP  , 000013 , 000000 , 000000)
                 053377    MOVEI   15, 000013
                 053400    POPJ    17, 000000
   QUADS TRANSLATED =   3

FUSED BLOCKS   8 THRU   10

MOVING BLOCK   8 TO 053401
       032224    (JUMP  , 000011 , 000000 , 000000)

MOVING BLOCK   9 TO 053411
       032266    (BLE   , 052376 , 032341 , 000011)
                 053432    CAMG    05, 032341
                 053433    JRST    00, 053421
       032270    (JUMP  , 000012 , 000000 , 000000)

MOVING BLOCK   10 TO 053434
       032274    (BLE   , 033661 , 032341 , 000010)
                 053437    MOVEI   15, 000010
                 053440    CAMG    04, 032341
                 053441    POPJ    17, 000000
       032276    (JUMP  , 000013 , 000000 , 000000)
```

```
                053442    MOVEI   15, 000013
                053443    POPJ    17, 000000

CODE MOTION ON SEGMENT   8

CSE ON BLOCK   8
   QUADS ELIMINATED =   0

CSE ON BLOCK  10
   QUADS ELIMINATED =   0
   NO QUADS REMOVED

COMPILING BLOCK   8
        032212    (MPY    , 033661 , 032341 , 032530)
                053444    MOVE    04, 033661
                053445    IMULI   04, 000062
        032214    (ADD    , 032530 , 033660 , 032531)
                053446    ADD     04, 033660
        032216    (ADD    , 032531 , 032345 , 032532)
        032220    (REPL   , 032346 , 000000 , 032532)
                053447    SETZM   00, 045607(04)
        032222    (REPL   , 032340 , 000000 , 052376)
                053450    MOVEI   04, 000001
                053451    MOVEM   04, 052376
        032224    (JUMP   , 000011 , 000000 , 000000)
   QUADS COMPILED =   6

MOVING BLOCK   9 TO 053452
        032266    (BLE    , 052376 , 032341 , 000011)
                053473    CAMG    05, 032341
                053474    JRST    00, 053462
        032270    (JUMP   , 000012 , 000000 , 000000)

COMPILING BLOCK  10
        032272    (ADD    , 033661 , 032340 , 033661)
                053475    AOS     04, 033661
        032274    (BLE    , 033661 , 032341 , 000010)
                053476    CAMG    04, 032341
                053477    JRST    00, 053444
        032276    (JUMP   , 000013 , 000000 , 000000)
                053500    MOVEI   15, 000013
                053501    POPJ    17, 000000
   QUADS COMPILED =   3

TRANSLATING BLOCK   7
        032206    (REPL   , 032340 , 000000 , 033661)
                053502    MOVEI   04, 000001
                053503    MOVEM   04, 033661
        032210    (JUMP   , 000010 , 000000 , 000000)
                053504    MOVEI   15, 000010
                053505    POPJ    17, 000000
   QUADS TRANSLATED =   2

TRANSLATING BLOCK  11
        032300    (ADD    , 033660 , 032340 , 033660)
                053506    MOVEI   04, 000001
                053507    ADD     04, 033660
                053510    MOVEM   04, 033660
        032302    (BLE    , 033660 , 032341 , 000007)
                053511    MOVEI   15, 000007
                053512    CAMG    04, 032341
                053513    POPJ    17, 000000
        032304    (JUMP   , 000014 , 000000 , 000000)
                053514    MOVEI   15, 000014
                053515    POPJ    17, 000000
   QUADS TRANSLATED =   3

FUSED BLOCKS   7 THRU  11

MOVING BLOCK   7 TO 053516
        032210    (JUMP   , 000010 , 000000 , 000000)

MOVING BLOCK   8 TO 053520
        032224    (JUMP   , 000011 , 000000 , 000000)

MOVING BLOCK   9 TO 053526
        032266    (BLE    , 052376 , 032341 , 000011)
                053547    CAMG    05, 032341
                053550    JRST    00, 053536
        032270    (JUMP   , 000012 , 000000 , 000000)

MOVING BLOCK  10 TO 053551
        032274    (BLE    , 033661 , 032341 , 000010)
                053552    CAMG    04, 032341
                053553    JRST    00, 053520
        032278    (JUMP   , 000013 , 000000 , 000000)

MOVING BLOCK  11 TO 053554
        032302    (BLE    , 033660 , 032341 , 000007)
                053557    MOVEI   15, 000007
                053560    CAMG    04, 032341
                053561    POPJ    17, 000000
        032304    (JUMP   , 000014 , 000000 , 000000)
                053562    MOVEI   15, 000014
                053563    POPJ    17, 000000

CODE MOTION ON SEGMENT   7

CSE ON BLOCK   7
   QUADS ELIMINATED =   0

CSE ON BLOCK  11
   QUADS ELIMINATED =   0
   NO QUADS REMOVED

COMPILING BLOCK   7
        032206    (REPL   , 032340 , 000000 , 033661)
                053564    MOVEI   04, 000001
                053565    MOVEM   04, 033661
        032210    (JUMP   , 000010 , 000000 , 000000)
   QUADS COMPILED =   2

MOVING BLOCK   8 TO 053566
        032224    (JUMP   , 000011 , 000000 , 000000)

MOVING BLOCK   9 TO 053574
        032266    (BLE    , 052376 , 032341 , 000011)
                053615    CAMG    05, 032341
                053616    JRST    00, 053604
        032270    (JUMP   , 000012 , 000000 , 000000)

MOVING BLOCK  10 TO 053617
        032274    (BLE    , 033661 , 032341 , 000010)
                053620    CAMG    04, 032341
                053621    JRST    00, 053566
        032276    (JUMP   , 000013 , 000000 , 000000)

COMPILING BLOCK  11
        032300    (ADD    , 033660 , 032340 , 033660)
                053622    AOS     04, 033660
        032302    (BLE    , 033660 , 032341 , 000007)
                053623    CAMG    04, 032341
                053624    JRST    00, 053564
        032304    (JUMP   , 000014 , 000000 , 000000)
                053625    MOVEI   15, 000014
                053626    POPJ    17, 000000
   QUADS COMPILED =   3
```

B.2 The Linear Equation Solver: LES

```
1.    C     MATRIX INVERSION USING A LINEAR EQUATION SOLVER
2.    C
3.    C     REFS: 1) ALGORITHM 423(CACM 15,4(APRIL 1972),274)
4.    C           2) FORSYTHE,G.E., AND MOLER,C.B. "COMPUTER SOLUTION
5.    C              OF LINEAR ALGEBRAIC SYSTEMS", PRENTICE-HALL,
6.    C              ENGLEWOOD CLIFFS,N.J.,1967.
7.    C           3) GREGORY,R.T., AND KARNEY,D.L., "A COLLECTION OF
8.    C              MATRICES FOR TESTING COMPUTATIONAL ALGORITHMS",
9.    C              WILLEY-INTERSCIENCE, NEW YORK,1969.
10.   C
11.   C     SUBROUTINES USED ARE NOT THOSE GIVEN IN THE TEXTBOOK, BUT
12.   C     THE REPLACEMENTS GIVEN BY MOLER IN ALGORITHM 423.
13.   C
14.         REAL A(100,100),B(100)
15.         INTEGER IP(100)
16.         READ(1,2)N
17.   2     FORMAT(6X,I2)
18.         TYPE 3,N
19.   3     FORMAT(' N = ',I2)
20.         NDIM=100
21.   C     GENERATE TEST MATRIX A: A(I,J)=N-ABS(I-J)
22.   C     SEE [3], EXAMPLE 7.6
23.   C
24.         DO 1 I=1,N
25.         DO 1 J=1,N
26.         A(I,J)=N-IABS(I-J)
27.   1     A(J,I)=A(I,J)
28.   C
29.   C     MAIN PROGRAM
30.         CALL DECOMP(N,NDIM,A,IP)
31.         IF(IP(N) .NE. 0)GO TO 30
32.         TYPE 40
33.   40    FORMAT(' MATRIX SINGULAR')
34.         STOP
35.   30    DO 10 J=1,N
36.         DO 20 I=1,N
37.   20    B(I)=0.0
38.         B(J)=1.0
39.   C     THE JTH CALL PRODUCES IN B THE JTH COLUMN OF THE INVERSE
40.   10    CALL SOLVE(N,NDIM,A,B,IP)
41.         END
42.   C
43.         SUBROUTINE DECOMP(N,NDIM,A,IP)
44.         REAL A(NDIM,NDIM),T
45.         INTEGER IP(NDIM)
46.         IP(N)=1
47.         DO 6 K=1,N
48.         IF(K .EQ. N)GO TO 5
49.         KP1=K+1
50.         M=K
51.         DO 1 I=KP1,N
52.         IF(ABS(A(I,K)) .GT. ABS(A(M,K)))M=I
53.   1     CONTINUE
54.         IP(K)=M
55.         IF(M .NE. K)IP(N)=-IP(N)
56.         T=A(M,K)
57.         A(M,K)=A(K,K)
58.         A(K,K)=T
59.         IF(T .EQ. 0.)GO TO 5
60.         DO 2 I=KP1,N
61.   2     A(I,K)=-A(I,K)/T
62.         DO 4 J=KP1,N
63.         T=A(M,J)
64.         A(M,J)=A(K,J)
65.         A(K,J)=T
66.         IF(T .EQ. 0.)GO TO 4
67.         DO 3 I=KP1,N
68.   3     A(I,J)=A(I,J)+A(I,K)*T
69.   4     CONTINUE
70.   5     IF(A(K,K) .EQ. 0.)IP(N)=0
71.   6     CONTINUE
72.         RETURN
73.         END
74.   C
75.         SUBROUTINE SOLVE(N,NDIM,A,B,IP)
76.         REAL A(NDIM,NDIM),B(NDIM),T
77.         INTEGER IP(NDIM)
78.         IF(N .EQ. 1)GO TO 9
79.         NM1=N-1
80.         DO 7 K=1,NM1
81.         KP1=K+1
82.         M=IP(K)
83.         T=B(M)
84.         B(M)=B(K)
85.         B(K)=T
86.         DO 7 I=KP1,N
87.   7     B(I)=B(I)+A(I,K)*T
88.         DO 8 KB=1,NM1
89.         KM1=N-KB
90.         K=KM1+1
91.         B(K)=B(K)/A(K,K)
92.         T=-B(K)
93.         DO 8 I=1,KM1
94.   8     B(I)=B(I)+A(I,K)*T
95.   9     B(1)=B(1)/A(1,1)
96.         RETURN
97.         END
```

B.3 A Prime Number Generator: SIEVE2

```
      C     REF: CACM 10.9(SEPT.,1967), P. 570
      C     ALGORITHM 311: PRIME NUMBER GENERATOR 2
      C        THE ALGORITM HAS BEEN MODIFIED TO GENERATE THE
      C        FIRST K PRIMES INSTEAD OF THE PRIMES 9M.
      C
      C     NOTE: THERE ARE 78,498 PRIMES LESS THAN 10**6.
      C
      C     USING M=10**6 AS AN UPPER BOUND, AN UPPER BOUND OF 200
      C     WILL SUFFICE FOR THE ARRAYS Q,DQ,SQ,R AND RR
      C     (I.E., 2.7*SQRT(10**6)/LN(10**6,)=200)
            INTEGER Q(200),DQ(200),SQ(200),R(200),RR(200)
            INTEGER P(25000)
            INTEGER I,JJ,JX,KK,N,JR,DN
            LOGICAL T
            P(1)=2
            DN=2
            P(2)=3
            J=3
            JJ=3
            K=3
            R(3)=3
            P(3)=5
            Q(3)=25
            DQ(3)=10
            SQ(3)=30
            READ(1,2)KK
      2     FORMAT(6X,I5)
            TYPE 3,KK
      3     FORMAT(' K = ',I5)
            N=7
      10    T=.TRUE.
            DN=6-DN
            DO 20 I=3,JJ
            IR=R(I)
            IF(N .NE. Q(IR))GO TO 20
            Q(IR)=N+DQ(IR)
            DQ(IR)=SQ(IR)-DQ(IR)
            T=.FALSE.
            IF(I .NE. JJ)GO TO 20
            JJ=JJ+1
            IF(IR .NE. J)GO TO 20
            J=J+1
            R(J)=J
            Q(J)=P(J)*P(J)
            SQ(J)=6*P(J)
            DQ(J)=SQ(J)*(1+(P(J)/3))-2*Q(J)
      20    CONTINUE
            IF(.NOT. T)GO TO 80
            K=K+1
            P(K)=N
            IF(K .EQ. KK)STOP
      30    IF(JJ .EQ. 3)GO TO 80
            JJ=JJ-1
            IF(Q(R(JJ)) .LT. Q(R(JJ-1)))GO TO 30
      C     SIFT SORT
            RR(3)=R(3)
            IF(JJ .LT. 4)GO TO 90
            DO 110 IR=4,JJ
            I=IR-1
      40    IF(Q(R(IR)) .GE. Q(RR(I)))GO TO 110
            RR(I+1)=RR(I)
            I=I-1
            IF(I .GE. 3)GO TO 40
      110   RR(I+1)=R(IR)
      C     MERGE SORT
      90    I=3
            IR=3
            JR=JJ+1
      50    IF(Q(RR(IR)) .GT. Q(R(JR)))GO TO 120
            R(I)=RR(IR)
            IR=IR+1
            IF(IR .GT. JJ)GO TO 70
            GO TO 130
      120   R(I)=R(JR)
            JR=JR+1
            IF(JR .GT. J)GO TO 60
      130   I=I+1
            GO TO 50
      60    I=I+1
            R(I)=RR(IR)
            IR=IR+1
            IF(IR .LE. JJ)GO TO 60
      70    JJ=3
      80    N=N+DN
            GO TO 10
            END
```

B.4 A Student Electrical Engineering Problem: EE

```
      REAL K1,K2
      C1=96E-6
      VB=120
      R=6800
      C2L=1.
      C2U=100.
      C2I=1.
      C2L=C2L*1E-6
      C2U=C2U*1E-6
      C2I=C2I*1E-6
      C3=1.
      C3U=100.
      READ(1,2)C3I
2     FORMAT(6X,F3.1)
      TYPE 3,C3I
3     FORMAT(' C3I = ',F3.1)
      C3=C3*1E-6
      C3U=C3U*1E-6
      C3I=C3I*1E-6
      C2=C2L
      UMAX=0
      VMAX=0
      C3BEST=0
      C2BEST=0
40    A=-(2*C2+C1)/(2*R*C1*C2)
      B=SQRT(4*C2*C2+C1*C1)/(2*R*C1*C2)
      Q20=VB/(1/C1+1/C2+1/C3)
      K2=1/(B+B)*((A+B)*Q20-VB/R+Q20/R*(1/C1+1/C2))
      K1=Q20-K2
      BASE=-K1*(A+B)/(K2*(A-B))
      Q2TP=K1*(BASE)**((A+B)*(-1/(B+B)))+K2*BASE**((A-B)*(-1/(B+B)))
      V0=120-Q2TP/C2+Q20/C3
      U=.5*(C1*VB*VB+Q2TP*Q2TP/C2+Q20*Q20/C3)
      IF(C2 .GT. C2U)GO TO 30
      IF(U .LE. UMAX)GO TO 50
      IF(V0 .LT. 240)GO TO 50
      VMAX=V0
      UMAX=U
      C3BEST=C3
      C2BEST=C2
50    C2=C2+C2I
      GO TO 40
30    C2=C2L
      IF(C3 .GT. C3U)GO TO 100
      C3=C3+C3I
      GO TO 40
100   C2BEST=C2BEST*1E6
      C3BEST=C3BEST*1E6
      TYPE 1,C2BEST,C3BEST,VMAX,UMAX
1     FORMAT(6X,'C2',11X,'C3',9X,'V0',8X,'WATTS',/,4F12.5)
      END
```

B.5 A Generalized Eigenvalue Problem: QZ

```
C      QZ ALGORITHM
C      REF: MOLER,C.B., AND STEWART,G.W., "AN ALGORITHM FOR
C           THE GENERALIZED MATRIX EIGENVALUE PROBLEMS",
C           SIAM J. NUMER. ANAL. 9,4(DEC. 1972).
C
       DIMENSION A(50,50),B(50,50),X(50,50),AR(50),
      1          ALFR(50),ALFI(50),BETA(50),IT(50)
C      DIMENSION RAM(50)
       READ(1,2000)N
 2000  FORMAT(6X,I2)
       TYPE 2001,N
 2001  FORMAT(' N = ',I2)
C      GENERATE TEST DATA
       DO 20 I=1,N
         AR(I)=I
         BT(I)=N-I+1
         DO 10 J=1,N
           A(I,J) = AR(I)
           B(I,J) = BT(I)
           X(I,J) = 0.
   10    CONTINUE
         A(I,I) = 2.*A(I,I)
         B(I,I) = 2.*B(I,I)
   20  CONTINUE
       CALL QZ(5D,N,A,B,1.E-8,AR,ALFR,ALFI,BETA,IT,-TRUE.,X)
C      DO 30 I=1,N
C        RAM(I) = AR(I)/BT(I)
C        PRINT 1001,RAM(I),AR(I),BT(I)
C        DO 40 J=1,N,5
C  40      PRINT 1001,X(J,I),X(J+1,I),X(J+2,I),X(J+3,I),X(J+4,I)
C1001    FORMAT(' ',5E12.4)
C   30   CONTINUE
       STOP
       END
C
       SUBROUTINE QZ(ND,N,A,B,EPS,ALFR,ALFI,BETA,ITER,
      1              WANTX,X)
       DIMENSION A(ND,ND),B(ND,ND),ALFR(N),ALFI(N),BETA(N),
      1          X(ND,ND),ITER(N)
       LOGICAL WANTX
       CALL QZHES(ND,N,A,B,WANTX,X)
       CALL QZIT(ND,N,A,B,EPS,EPSA,EPSB,ITER,WANTX,X)
       CALL QZVAL(ND,N,A,B,EPSB,ALFR,ALFI,BETA,WANTX,X)
       IF(WANTX) CALL QZVEC(ND,N,A,B,EPSA,EPSB,ALFR,ALFI,
      1BETA,X)
       RETURN
       END
C
       SUBROUTINE QZHES(ND,N,A,B,WANTX,X)
       DIMENSION A(ND,ND),B(ND,ND),X(ND,ND)
       LOGICAL WANTX
       IF(.NOT.WANTX) GO TO 10
       DO 3 I=1,N
         DO 2 J=1,N
           X(I,J) = 0
    2    CONTINUE
         X(I,I) = 1.
    3  CONTINUE
   10  NM1 = N-1
       DO 100 L=1,NM1
         L1 = L+1
         S = 0.
         DO 20 I=L,N
           IF(ABS(B(I,L)) .GT. S) S = ABS(B(I,L))
   20    CONTINUE
         IF(S .EQ. 0.) GO TO 100
         IF(ABS(B(L,L)) .GT. S) S = ABS(B(L,L))
         R = 0.
         DO 25 I=L,N
           B(I,L) = B(I,L)/S
           R = R + B(I,L)**2
   25    CONTINUE
         R = SQRT(R)
         IF(B(L,L) .LT. 0.) R = -R
         B(L,L) = B(L,L) + R
         RHO = R*B(L,L)
         DO 50 J=L1,N
           T = 0.
           DO 30 I=L,N
             T = T + B(I,L)*B(I,J)
   30      CONTINUE
           T = -T/RHO
           DO 40 I=L,N
             B(I,J) = B(I,J) + T*B(I,L)
   40      CONTINUE
   50    CONTINUE
         DO 80 J=1,N
           T = 0.
           DO 60 I=L,N
             T = T + B(I,L)*A(I,J)
   60      CONTINUE
           T = -T/RHO
           DO 70 I=L,N
             A(I,J) = A(I,J) + T*B(I,L)
   70      CONTINUE
   80    CONTINUE
         B(I,L) = -S*R
         DO 90 I=L1,N
           B(I,L) = 0.
   90    CONTINUE
  100  CONTINUE
       IF(N .LE. 2) GO TO 170
       NM2 = N-2
       DO 160 K=1,NM2
         K1 = K+1
         NK1 = N-K-1
         DO 150 LB=1,NK1
           L = N-LB
           L1 = L+1
           CALL HSH2(A(L,K),A(L1,K),U1,U2,V1,V2)
           IF(U1 .NE. 1.) GO TO 125
           DO 110 J=K,N
           T = A(L,J) + U2*A(L1,J)
           A(L,J) = A(L,J) + T*V1
           A(L1,J) = A(L1,J) + T*V2
  110      CONTINUE
           A(L1,K) = 0.
           DO 120 J=L,N
           T = B(L,J) + U2*B(L1,J)
           B(L,J) = B(L,J) + T*V1
           B(L1,J) = B(L1,J) + T*V2
  120      CONTINUE
  125      CALL HSH2(B(L1,L1),B(L1,L),U1,U2,V1,V2)
           IF(U1 .NE. 1.) GO TO 150
           DO 130 I=1,L1
           T = B(I,L1) + U2*B(I,L)
           B(I,L1) = B(I,L1) + T*V1
           B(I,L) = B(I,L) + T*V2
  130      CONTINUE
           B(L1,L) = 0.
           DO 140 I=1,N
           T = A(I,L1) + U2*A(I,L)
           A(I,L1) = A(I,L1) + T*V1
           A(I,L) = A(I,L) + T*V2
  140      CONTINUE
           IF(.NOT. WANTX) GO TO 150
           DO 145 I=1,N
           T = X(I,L1) + U2*X(I,L)
           X(I,L1) = X(I,L1) + T*V1
           X(I,L) = X(I,L) + T*V2
  145      CONTINUE
  150    CONTINUE
  160  CONTINUE
  170  CONTINUE
       RETURN
       END
C
       SUBROUTINE QZIT(ND,N,A,B,EPS,EPSA,EPSB,ITER,WANTX,X)
       DIMENSION A(ND,ND),B(ND,ND),X(ND,ND)
       DIMENSION ITER(N)
       LOGICAL WANTX,NOD
       ANORM = 0.
       BNORM = 0.
       DO 185 I=1,N
```

```
      ITER(I) = 0
      ANI = 0.
      IF(I .NE. 1) ANI = ABS(A(I,I-1))
      BNI = 0.
      DO 180 J=I,N
        ANI = ANI + ABS(A(I,J))
        BNI = BNI + ABS(B(I,J))
180   CONTINUE
      IF(ANI .GT. ANORM) ANORM = ANI
      IF(BNI .GT. BNORM) BNORM = BNI
185 CONTINUE
    EPSA = EPS*ANORM
    EPSB = EPS*BNORM
    M = N
200 IF(M .LE. 2) GO TO 390
    DO 220 LB=1,M
      L =M-LB+1
      IF(L .EQ. 1) GO TO 260
      IF(ABS(A(L,L-1)) .LE. EPSA) GO TO 230
220 CONTINUE
230 A(L,L-1) = 0.
    IF(L .LT. M-1) GO TO 260
    M = L-1
    GO TO 200
260 IF(ABS(B(L,L)) .GT. EPSB) GO TO 300
    B(L,L) = 0.
    L1 = L+1
    CALL HSH2(A(L,L),A(L,L1),U1,U2,V1,V2)
    IF(U1 .NE. 1.) GO TO 280
    DO 270 J=L,N
      T = A(L,J) + U2*A(L1,J)
      A(L,J) = A(L,J) + T*V1
      A(L1,J) = A(L1,J) + T*V2
      T = B(L,J) + U2*B(L1,J)
      B(L,J) = B(L,J) +T*V1
      B(L1,J) = B(L1,J) + T*V2
270 CONTINUE
280 L = L1
    GO TO 230
300 M1 = M-1
    L1 = L+1
    CONST = 0.75
    ITER(M) = ITER(M) + 1
    IF(ITER(M) .EQ. 1) GO TO 305
    IF(ABS(A(M,M-1)) .LT. CONST*OLD1) GO TO 305
    IF(ABS(A(M-1,M-2)) .LT. CONST*OLD2) GO TO 305
    IF(ITER(M) .EQ. 10) GO TO 310
    IF(ITER(M) .GT. 30) GO TO 380
305 B11 = B(L,L)
    B22 = B(L1,L1)
    IF(ABS(B22) .LT. EPSB) B22 = EPSB
    B33 = B(M1,M1)
    IF(ABS(B33) .LT. EPSB) B33 = EPSB
    B44 = B(M,M)
    IF(ABS(B44) .LT. EPSB) B44 = EPSB
    A11 = A(L,L)/B11
    A12 = A(L,L1)/B22
    A21 = A(L1,L)/B11
    A22 = A(L1,L1)/B22
    A33 = A(M1,M1)/B33
    A34 = A(M1,M)/B44
    A43 = A(M,M1)/B33
    A44 = A(M,M)/B44
    B12 = B(L,L1)/B22
    B34 = B(M1,M)/B44
    A10 = ((A33-A11)*(A44-A11) - A34*A43 + A43*B34*A11)/A21
   1      + A12 - A11*B12
    A20 = (A22-A11-A21*B12) - (A33-A11) - (A44-A11) + A43*B34
    A30 = A(L+2,L1)/B22
    GO TO 315
310 A10 = 0.
    A20 = 0.
    A30 = 1.1605
315 OLD1 = ABS(A(M,M-1))
    OLD2 = ABS(A(M-1,M-2))
    IF(.NOT.WANTX) LOR1 = L
    IF(WANTX) LOR1 = 1
    IF(.NOT.WANTX) MORN = M
    IF(WANTX) MORN = N
    DO 360 K=L,M1
      MIO = K.NE.M1
      K1 = K+1
      K2 = K+2
      K3 = K+3
      IF(K3 .GT. M) K3 =M
      KM1 = K-1
      IF(KM1 .LT. L) KM1 =L
      IF(K .EQ. L) CALL HSH3(A10,A20,A30,U1,U2,U3,V1,V2,V3)
      IF(K.GT.L .AND. K.LT.M1)
   1    CALL HSH3(A(K,KM1),A(K1,KM1),A(K2,KM1),U1,U2,U3,V1,V2,V3)
      IF(K .EQ. M1) CALL HSH2(A(K,KM1),A(K1,KM1),U1,U2,V1,V2)
      IF(U1 .NE. 1.) GO TO 325
      DO 320 J=KM1,MORN
        T = A(K,J) + U2*A(K1,J)
        IF(MIO) T = T + U3*A(K2,J)
        A(K,J) = A(K,J) + T*V1
        A(K1,J) = A(K1,J) + T*V2
        IF(MIO) A(K2,J) = A(K2,J) + T*V3
        T = B(K,J) + U2*B(K1,J)
        IF(MIO) T = T + U3*B(K2,J)
        B(K,J) = B(K,J) + T*V1
        B(K1,J) = B(K1,J) + T*V2
        IF(MIO) B(K2,J) = B(K2,J) + T*V3
320   CONTINUE
      IF(K .EQ. L) GO TO 325
      A(K1,K-1) = 0.
      IF(MIO) A(K2,K-1) = 0.
325   IF(K .EQ. M1) GO TO 340
      CALL HSH3(B(K2,K2),B(K2,K1),B(K2,K),U1,U2,U3,V1,V2,V3)
      IF(U1 .NE. 1.) GO TO 340
      DO 330 I=LOR1,K3
        T = A(I,K2) + U2*A(I,K1) + U3*A(I,K)
        A(I,K2) = A(I,K2) + T*V1
        A(I,K1) = A(I,K1) + T*V2
        A(I,K) = A(I,K) + T*V3
        T = B(I,K2) + U2*B(I,K1) + U3*B(I,K)
        B(I,K2) = B(I,K2) + T*V1
        B(I,K1) = B(I,K1) + T*V2
        B(I,K) = B(I,K) + T*V3
330   CONTINUE
      B(K2,K) = 0.
      B(K2,K1) = 0.
      IF(.NOT.WANTX) GO TO 340
      DO 335 I=1,N
        T = X(I,K2) + U2*X(I,K1) + U3*X(I,K)
        X(I,K2) = X(I,K2) + T*V1
        X(I,K1) = X(I,K1) + T*V2
        X(I,K) = X(I,K) +  T*V3
335   CONTINUE
340   CALL HSH2(B(K1,K1),B(K1,K),U1,U2,V1,V2)
      IF(U1 .NE. 1.) GO TO 360
      DO 350 I=LOR1,K3
        T = A(I,K1) + U2*A(I,K)
        A(I,K1) = A(I,K1) + T*V1
        A(I,K) = A(I,K) + T*V2
        T = B(I,K1) + U2*B(I,K)
        B(I,K1) = B(I,K1) + T*V1
        B(I,K) = B(I,K) + T*V2
350   CONTINUE
      B(K1,K) = 0.
      IF(.NOT.WANTX) GO TO 360
      DO 355 I=1,N
        T = X(I,K1) + U2*X(I,K)
        X(I,K1) = X(I,K1) + T*V1
        X(I,K) = X(I,K) + T*V2
355   CONTINUE
360 CONTINUE
    GO TO 200
380 DO 385 I=1,M
      ITER(I) = -1
385 CONTINUE
390 CONTINUE
    RETURN
    END
```

```
      C
            SUBROUTINE QZVAL(NO,N,A,B,EPSB,ALFR,ALFI,BETA,WANTX,X)
            DIMENSION A(NO,NO),B(NO,NO),ALFR(N),ALFI(N),BETA(N),X(NO,NO)
            LOGICAL WANTX,FLIP
            M = N
         400 CONTINUE
               IF(M .EQ. 1) GO TO 410
               IF(A(M,M-1) .NE. 0.) GO TO 420
         410   ALFR(M) = A(M,M)
               BETA(M) = B(M,M)
               ALFI(M) = 0.
               M = M-1
               GO TO 490
         420   L = M-1
               IF(ABS(B(L,L)) .GT. EPSB) GO TO 425
                  B(L,L) = 0.
                  CALL HSH2(A(L,L),A(M,L),U1,U2,V1,V2)
                  GO TO 460
         425   IF(ABS(B(M,M)) .GT. EPSB) GO TO 430
                  B(M,M) = 0.
                  CALL HSH2(A(M,M),A(M,L),U1,U2,V1,V2)
                  BN = 0.
                  GO TO 435
         430   AN = ABS(A(L,L)) + ABS(A(L,M)) + ABS(A(M,L)) + ABS(A(M,M))
               BN = ABS(B(L,L)) + ABS(B(L,M)) + ABS(B(M,M))
                  A11 = A(L,L)/AN
               A12 = A(L,M)/AN
               A21 = A(M,L)/AN
               A22 = A(M,M)/AN
               B11 = B(L,L)/BN
               B12 = B(L,M)/BN
               B22 = B(M,M)/BN
               C = (A11*B22 + A22*B11 - A21*B12)/2.
               D = (A22*B11 - A11*B22 - A21*B12)**2/4.
              1      + A21*B22*(A12*B11 - A11*B12)
               IF(D .LT. 0.) GO TO 480
               IF(C .GE. 0.) E = (C + SQRT(D))/(B11*B22)
               IF(C .LT. 0.) E = (C - SQRT(D))/(B11*B22)
               A11 = A11 - E*B11
               A12 = A12 - E*B12
               A22 = A22 - E*B22
               FLIP = (ABS(A11)+ABS(A12)) .GE. (ABS(A21)+ABS(A22))
               IF(FLIP) CALL HSH2(A12,A11,U1,U2,V1,V2)
               IF(.NOT.FLIP) CALL HSH2(A22,A21,U1,U2,V1,V2)
         435   IF(U1 .NE. 1.) GO TO 450
               DO 440 I=1,M
                  T = A(I,M) + U2*A(I,L)
                  A(I,M) = A(I,M) + V1*T
                  A(I,L) = A(I,L) + V2*T
                  T = B(I,M) + U2*B(I,L)
                  B(I,M) = B(I,M) + V1*T
                  B(I,L) = B(I,L) + V2*T
         440   CONTINUE
               IF(.NOT.WANTX) GO TO 450
               DO 445 I=1,N
                  T = X(I,M) + U2*X(I,L)
                  X(I,M) = X(I,M) + V1*T
                  X(I,L) = X(I,L) + V2*T
         445   CONTINUE
         450   IF(BN .EQ. 0.) GO TO 475
               FLIP = AN .GE. ABS(E)*BN
               IF(FLIP) CALL HSH2(B(L,L),B(M,L),U1,U2,V1,V2)
               IF(.NOT.FLIP) CALL HSH2(A(L,L),A(M,L),U1,U2,V1,V2)
         460   IF(U1 .NE. 1.) GO TO 475
               DO 470 J=L,N
                  T = A(L,J) + U2*A(M,J)
                  A(L,J) = A(L,J) + V1*T
                  A(M,J) = A(M,J) + V2*T
                  T = B(L,J) + U2*B(M,J)
                  B(L,J) = B(L,J) + V1*T
                  B(M,J) = B(M,J) + V2*T
         470   CONTINUE
         475   A(M,L) = 0.
               B(M,L) = 0.
               ALFR(L) = A(L,L)
               ALFR(M) = A(M,M)
               BETA(L) = B(L,L)
               BETA(M) = B(M,M)
               ALFI(M) = 0.
               ALFI(L) = 0.
               M = M-2
               GO TO 490
         480   ER = C/(B11*B22)
               EI = SQRT(-D)/(B11*B22)
               A11R = A11 - ER*B11
               A11I = EI*B11
               A12R = A12 - ER*B12
               A12I = EI*B12
               A21R = A21
               A21I = 0.
               A22R = A22 - ER*B22
               A22I = EI*B22
               FLIP = (ABS(A11R)+ABS(A11I)+ABS(A12R)+ABS(A12I)) .GE.
              1          (ABS(A21R)+ABS(A22R)+ABS(A22I))
               IF(FLIP) CALL CHSH2(A12R,A12I,-A11R,-A11I,CZ,SZR,SZI)
                  IF(.NOT.FLIP) CALL CHSH2(A22R,A22I,-A21R,-A21I,CZ,SZR,SZI)
               FLIP = AN .GE. (ABS(ER)+ABS(EI))*BN
               IF(FLIP) CALL CHSH2(CZ*B11+SZR*B12,SZI*B12,
              1          SZR*B22,SZI*B22,CQ,SQR,SQI)
               IF(.NOT.FLIP) CALL CHSH2(CZ*A11+SZR*A12,SZI*A12,
              1       CZ*A21+SZR*A22,SZI*A22,CQ,SQR,SQI)
               SSR = SQR*SZR + SQI*SZI
               SSI = SQR*SZI - SQI*SZR
               TR = CQ*CZ*A11 + CQ*SZR*A12 + SQR*CZ*A21 + SSR*A22
               TI = CQ*SZI*A12 - SQI*CZ*A21 + SSI*A22
               BDR = CQ*CZ*B11 + CQ*SZR*B12 + SSR*B22
               BDI = CQ*SZI*B12 + SSI*B22
               R = SQRT(BDR*BDR + BDI*BDI)
               BETA(L) = BN*R
               ALFR(L) = AN*(TR*BDR + TI*BDI)/R
               ALFI(L) = AN*(TR*BDI - TI*BDR)/R
               TR = SSR*A11 - SQR*CZ*A12 - CQ*SZR*A21 + CQ*CZ*A22
               TI = -SSI*A11 - SQI*CZ*A12 + CQ*SZI*A21
               BDR = SSR*B11 - SQR*CZ*B12 + CQ*CZ*B22
               BDI = -SSI*B11 - SQI*CZ*B12
               R = SQRT(BDR*BDR + BDI*BDI)
               BETA(M) = BN*R
               ALFR(M) = AN*(TR*BDR + TI*BDI)/R
               ALFI(M) = AN*(TR*BDI - TI*BDR)/R
               M = M-2
         490 IF(M .GT. 0) GO TO 400
            RETURN
            END
      C
            SUBROUTINE QZVEC(NO,N,A,B,EPSA,EPSB,ALFR,ALFI,BETA,X)
            DIMENSION A(NO,NO),B(NO,NO),ALFR(N),ALFI(N),BETA(N),X(NO,NO)
            LOGICAL FLIP
            M = N
         500 CONTINUE
               IF(ALFI(M) .NE. 0.) GO TO 550
               ALFM = ALFR(M)
               BETM = BETA(M)
               IF(ABS(ALFM) .LT. EPSA) ALFM = 0.
               IF(ABS(BETM) .LT. EPSB) BETM = 0.
               B(M,M) = 1.
               L = M-1
               IF(L .EQ. 0) GO TO 540
         510   CONTINUE
                  L1 = L+1
                  SL = 0.
                  DO 515 J=L1,M
               SL = SL + (BETM*A(L,J)-ALFM*B(L,J))*B(J,M)
         515      CONTINUE
                  IF(L .EQ. 1) GO TO 520
                  IF(A(L,L-1) .NE. 0.) GO TO 530
         520      D = BETM*A(L,L) - ALFM*B(L,L)
                  IF(D .EQ. 0.) D = (EPSA+EPSB)/2.
                  B(L,M) = -SL/D
                  L = L-1
                  GO TO 540
         530      K = L-1
                  SK = 0.
                  DO 535 J=L1,M
               SK = SK + (BETM*A(K,J)-ALFM*B(K,J))*B(J,M)
```

```
      535   CONTINUE
            TKK = BETM*A(K,K) - ALFM*B(K,K)
            TKL = BETM*A(K,L) - ALFM*B(K,L)
            TLK = BETM*A(L,K)
            TLL = BETM*A(L,L) - ALFM*B(L,L)
            D = TKK*TLL - TKL*TLK
            IF(D .EQ. 0.) D = (EPSA+EPSB)/2.
            B(L,M) = (TLK*SK - TKK*SL)/D
            FLIP = ABS(TKK) .GE. ABS(TLK)
            IF(FLIP) B(K,M) = (SK - TKL*B(L,M))/TKK
            IF(.NOT.FLIP) B(K,M) = (SL - TLL*B(L,M))/TLK
            L = L-2
      540   IF(L .GT. 0) GO TO 510
            M = M-1
            GO TO 590
      550   ALMR = ALFR(M-1)
            ALMI = ALFI(M-1)
            BETM = BETA(M-1)
            MR = M-1
            MI = M
            B(M-1,MR) = ALMI*B(M,M)/(BETM*A(M,M-1))
            B(M-1,MI) = (BETM*A(M,M)-ALMR*B(M,M))/(BETM*A(M,M-1))
            B(M,MR) = 0.
            B(M,MI) = -1.
            L = M-2
            IF(L .EQ. 0) GO TO 585
      560   CONTINUE
              LI = L+1
              SLR = 0.
              SLI = 0.
              DO 565 J=LI,M
            TR = BETM*A(L,J) - ALMR*B(L,J)
            TI = -ALMI*B(L,J)
            SLR = SLR + TR*B(J,MR) - TI*B(J,MI)
            SLI = SLI + TR*B(J,MI) + TI*B(J,MR)
      565   CONTINUE
            IF(L .EQ. 1) GO TO 570
            IF(A(L,L-1) .NE. 0.) GO TO 575
      570   DR = BETM*A(L,L) - ALMR*B(L,L)
            DI = -ALMI*B(L,L)
            CALL CDIV(-SLR,-SLI,DR,DI,B(L,MR),B(L,MI))
            L = L-1
            GO TO 585
      575   K = L-1
            SKR = 0.
            SKI = 0.
            DO 580 J=L,M
            TR = BETM*A(K,J) - ALMR*B(K,J)
            TI = -ALMI*B(K,J)
            SKR = SKR + TR*B(J,MR) - TI*B(J,MI)
            SKI = SKI + TR*B(J,MI) + TI*B(J,MR)
      580   CONTINUE
            TKKR = BETM*A(K,K) - ALMR*B(K,K)
            TKKI = -ALMI*B(K,K)
            TKLR = BETM*A(K,L) - ALMR*B(K,L)
            TKLI = -ALMI*B(K,L)
            TLKR = BETM*A(L,K)
            TLKI = 0.
            TLLR = BETM*A(L,L) - ALMR*B(L,L)
            TLLI = -ALMI*B(L,L)
            DR = TKKR*TLLR - TKKI*TLLI - TKLR*TLKR
            DI = TKKR*TLLI + TKKI*TLLR - TKLI*TLKR
            IF(DR .EQ. 0. .AND. DI.EQ.0.) DR = (EPSA+EPSB)/2.
            CALL CDIV(TLKR*SKR-TKKI*SLR-TKKR*SLI,
           1          TLKR*SKI-TKKR*SLI-TKKI*SLR,
           2          DR,DI,B(L,MR),B(L,MI))
            FLIP = (ABS(TKKR)+ABS(TKKI)) .GE. ABS(TLKR)
            IF(FLIP) CALL CDIV(-SKR-TKLR*B(L,MR)+TKLI*B(L,MI),
           1                   -SKI-TKLR*B(L,MI)-TKLI*B(L,MR),
           2                   TKKR,TKKI,B(K,MR),B(K,MI))
            IF(.NOT.FLIP) CALL CDIV(-SLR-TLLR*B(L,MR)+TLLI*B(L,MI),
           1                        -SLI-TLLR*B(L,MI)-TLLI*B(L,MR),
           2                        TLKR,TLKI,B(K,MR),B(K,MI))
            L = L-2
      585   IF(L .GT. 0) GO TO 560
            M = M-2
      590 IF(M .GT. 0) GO TO 500
            M = N
      600 CONTINUE
            DO 620 I=1,N
              S = 0.
              DO 610 J=1,M
            S = S+ X(I,J)*B(J,M)
      610     CONTINUE
              X(I,M) = S
      620   CONTINUE
            M = M-1
          IF(M .GT. 0) GO TO 600
          M = N
      630 CONTINUE
            S = 0.
            IF(ALFI(M) .NE. 0.) GO TO 650
            DO 635 I=1,N
              R = ABS(X(I,M))
              IF(R .LT. S) GO TO 635
              S = R
              D = X(I,M)
      635   CONTINUE
            DO 640 I=1,N
              X(I,M) = X(I,M)/D
      640   CONTINUE
            M = M-1
            GO TO 690
      650   DO 655 I=1,N
              R = X(I,M-1)**2 + X(I,M)**2
              IF(R .LT. S) GO TO 655
              S = R
              DR = X(I,M-1)
              DI = X(I,M)
      655   CONTINUE
            DO 660 I=1,N
              CALL CDIV(X(I,M-1),X(I,M),DR,DI,X(I,M-1),X(I,M))
      660   CONTINUE
            M = M-2
      690 IF(M .GT. 0) GO TO 630
      700 RETURN
          END
C
          SUBROUTINE HSH3(A1,A2,A3,U1,U2,U3,V1,V2,V3)
          IF(A2.EQ.0. .AND. A3.EQ.0.) GO TO 10
          S = ABS(A1) + ABS(A2) + ABS(A3)
          U1 = A1/S
          U2 = A2/S
          U3 = A3/S
          R = SQRT(U1*U1+U2*U2+U3*U3)
          IF(U1 .LT. 0.) R = -R
          V1 = -(U1 + R)/R
          V2 = -U2/R
          V3 = -U3/R
          U1 = 1.
          U2 = V2/V1
          U3 = V3/V1
          RETURN
       10 U1 = 0.
          RETURN
          END
C
          SUBROUTINE HSH2(A1,A2,U1,U2,V1,V2)
          IF(A2 .EQ. 0.) GO TO 10
          S = ABS(A1) + ABS(A2)
          U1 = A1/S
          U2 = A2/S
          R = SQRT(U1*U1+U2*U2)
          IF(U1 .LT. 0.) R = -R
          V1 = -(U1 + R)/R
          V2 = -U2/R
          U1 = 1.
          U2 = V2/V1
          RETURN
       10 U1 = 0.
          RETURN
          END
C
          SUBROUTINE CHSH2(A1R,A1I,A2R,A2I,CS,SI)
```

```
      IF(A2R.EQ.0. .AND. A2I.EQ.0.) GO TO 10
      IF(A1R.EQ.0. .AND. A1I.EQ.0.) GO TO 20
      R = SQRT(A1R*A1R+A1I*A1I)
      C = R
      SR = (A1R*A2R+A1I*A2I)/R
      SI = (A1R*A2I-A1I*A2R)/R
      R = SQRT(C*C+SR*SR+SI*SI)
      C = C/R
      SR = SR/R
      SI = SI/R
      RETURN
   10 C = 1.
      SR = 0.
      SI = 0.
      RETURN
   20 C = 0.
      SR = 1.
      SI = 0.
      RETURN
      END
C
      SUBROUTINE CDIV(XR,XI,YR,YI,ZR,ZI)
      IF(ABS(YR) .LT. ABS(YI)) GO TO 10
      WR = XR/YR
      WI = XI/YR
      VI = YI/YR
      D = 1. + VI*VI
      ZR = (WR + WI*VI)/D
      ZI = (WI - WR*VI)/D
      RETURN
   10 WR = XR/YI
      WI = XI/YI
      VR = YR/YI
      D = VR*VR + 1.
      ZR = (WR*VR + WI)/D
      ZI = (WI*VR - WR)/D
      RETURN
      END
```

Appendix C

Code Matrices for Integer '+'

In order to show in detail the form of the analysis used by the "fair" code machine language generator for the arithmetic and logical operators (see Chapter 3, Section 3.1.3.5), this appendix contains the code matrices for integer addition. There are two code matrices; one for quads of the form (+,V,E,V) or (+,E,V,V) with the first two arguments commuted, and one for quads of the form (+,E1,E2,T), where E1, E2, and E are arguments that may be simple variables, parameters, results or indirect results; V is a simple varibale, parameter, or indirect result; and T is a temporary result. Each matrix contains 16 cases, depending on the mode of the operands. There are four possible operand modes: MEM, NUM, REG, and REG+NUM. Thus, for example, the case "MEM/REG(s)" means the first operand is in memory while the second is in register s.

The logic of a case analysis is presented in tabular form with the following conventions:

1) The machine language instructions generated are expressed in MACRO-10 [PDP71a], the assembly language for the PDP-10. Curly brackets are used for the conditional generation of information. Thus, for example,

```
MOVE    r,{*}E1
```

means to generate a MOVE instruction with r as the register field, address of E1 as the address field, and the indirect bit set if tag bit 1 of the '+' quad is set (see Appendix A, Section A.3).

2) The information that controls the analysis resides in fields in the temp or register table, or tag bits in the '+' quad. The mnemonics for the tags, along with their meaning, can be found in Appendix A, Section A.3. The mnemonics and their meanings for fields in the register and temp table are:

mnemonic	meaning
RANGE	range field of temp entry
NB	neg-bit field of temp entry
INFO	information field of temp entry
USES	use field of register entry

To reference fields in the tables, an indexing scheme is used with the name of the operand being the index. For example,

NB[E2] ← '+'

means set the neg-bit in the temp table entry for E2 (a temporary) to plus.

3) The following variables are used:

variable	meaning
Q	the address of the '+' quad.
r	an unused register. This register is allocated by the register allocation algorithm. Initially, the register has no associated temporary or variable.
s,t	registers containing the operands.
L	a literal constant, folded or otherwise.
C	address of a constant, folded or otherwise.

4) The following shorthand notation is used for table headings:

symbol	meaning
*	indirect tag bit of '+' quad set
+	neg-bit field of temp plus
-	neg-bit field of temp negative
temp	temporary tag bit of '+' quad set
cons	operand is a constant
lit	operand is a literal

As code is generated, fields in the temp and register tables must be updated. Only those updations pertinent to the clarification of the machine language generated are included in the logic. Some updations are given explicitly, while others are indicated by enclosing them in double quotes. The latter updations are:

update	meaning
"associate Reg with T" or "Reg←T"	update the correct register and temp table entry to reflect that Reg contains T.
"associate Reg with T+NUM" or "Reg←T+NUM"	update the correct register and temp table entry to reflect that Reg contains T which has mode = "REG+NUM".
"associate Num with T" or "NUM←T"	Num is a number which resides in a temp. Associate it with the result temp T by moving in the temp table the number indicator and information fields of the temp containing Num to the corresponding fields of T.
"negate E"	E is a temp. Perform the following modifications to its entry in the temp table: INFO[E] ← -INFO[E] NB[E] ← '+'

Subcases for a case are numbered using the Dewey decimal notation. For example, if k is the number of a case, k.1, ... , k.m are its subcases, k.1.1, ... , k.1.n are the sub-subcases of subcase k.1, etc. . The logic for a case is to be read sequentially with subcases being disjoint and code not appearing under any subcase being common to all subcases. Some cases are similar, and to avoid duplication, one case is transformed into another by taking certain actions. The actions are enclosed in single quotes and are:

action	meaning
'reset temp mode'	reset the mode of the temp from "REG" to "REG+NUM" with the number set to zero. If Q⇒RANGE[temp], then set the USES field of its associated register to 1.
'E1 ↔ E2'	interchange the attributes of the two operands.
'same as k'	the analysis is the same as for case k.

C.1 Code Matrix for (+,V,E,V) or (+E,V,V) Commuted

1. MEM/MEM

```
MOVE   r,{*}E
ADDB   r,{*}V
```

2. MEM/NUM

2.1 E a constant: same as (1)

2.2 E a literal

E=1	E≠1≠0	
AOS r,{*}V	not *	*
	MOVEI r,E ADDB r,{*}V	MOVE r,E

2.3 E a temp with mode="NUM"

folded cons	lit
same as (1)	same as (2.2)

3. MEM/REG(s)

3.1 E an indirect temp

'reset temp mode'
'same as (4)'

3.2 E a temp

Q<RANGE[E]				Q=RANGE[E]			
+		-		+		-	
ADDM s,{*}V		SUBM s,{*}V MOVNS {*}		ADDB s,{*}V		MOVNS s ADDB s,{*}V	

3.3 E not a temp

no temp associated with s	temp associated with s
ADDB s,{*}V	ADDM s,{*}V

4. MEM/REG(s)+NUM

4.1 Q<RANGE[E] or (Q=RANGE[E] and USES[s]>1)

temp				*temp	
+		-		+	-
MOVE r,s		MOVN r,s		——	MOVNS s "negate L"
cons	lit	cons	lit		
ADD r,C	ADDI r,L	SUB r,C	SUBI r,L	MOVE r,L(s) ADDB r,{*}V	

4.2 Q=RANGE[E] and USES[s]=1

temp				*temp
cons		lit		same as (4.1) with s replacing r
ADD s,C		ADDI s,L		
+		-		
ADDB s,{*}V		MOVNS s ADDB s,{*}V		

5. NUM/MEM (impossible)

6. NUM/NUM (impossible)

7. NUM/REG(s) (impossible)

8. NUM/REG(s)+NUM (impossible)

9. REG(s)/MEM

9.1 V an indirect temp

'reset temp mode'
'same as (13)'

9.2 V in a register

no temp associated with s	temp associated with s
ADD s,{*}E	MOVE r,{*}E
MOVEM s,{*}V	ADDB r,{*}V

10. REG(s)/NUM

10.1 V an indirect temp

'reset temp mode'
'same as (14)'

10.2 No temp associated with s

cons	lit		
ADD s,C MOVEM s,V	E=1	E≠1≠0	
	AOS s,V	not *	*
		ADDI s,L MOVEM s,L	ADD s,L

10.3 Temp associated with s: same as (2)

11. REG(t)/REG(s)

11.1 V an indirect temp

'reset temp mode'
'same as (15)'

11.2 V in a register: same as (3)

12. REG(t)/REG(s)+NUM

12.1 V an indirect temp

'reset temp mode'
'same as (16)'

12.2 Temp associated with t: same as (4)

12.3 no temp associated with t

temp				*temp	
+		-		+	-
ADD t,s		SUB t,s		—	MOVNS s "negate L"
cons	lit	cons	lit		
ADD t,C	ADDI t,L MOVEM t,V	SUB t,C	SUBI t,L	ADD t,L(s) MOVEM t,V	

For cases 13-16, V is a temp with mode="REG+NUM" where NUM is a literal. In order to use the literal as an index, it must be positive. Before code for the case is generated, the neg-bit is checked.

+	-
———	MOVNS V "negate L"

13. REG(s)+NUM/MEM

```
MOVE   r,{*}E
ADDM   r,L(s)
```

14. REG(s)+NUM/NUM

14.1 E a constant: same as (13)

14.2 E a literal

E=1	E≠1≠0	
AOS L(s)	not *	*
	MOVEI r,E ADDM r,L(s)	MOVE r,E

14.3 E a temp with mode="NUM"

folded cons	lit
same as (13)	same as (14.2)

15. REG(t)+NUM/REG(s)

15.1 E an indirect temp

```
'reset mode of temp'
'same as (16)'
```

15.2 E not a temp

```
ADDM   s,L(t)
```

15.3 E a temp

+	-
ADDM s,L(t)	SUBM s,L(t) MOVNS L(t)

16. REG(t)+NUM/REG(s)+NUM

16.1 Q=RANGE[E] and USES[s]=1

temp		*temp	
cons	lit	+	-
ADD s,C	ADDI s,L	—	MOVNS s "negate L"
+	-	MOVE s,L(s)	
—	MOVNS s		

ADDM s,L(t)

16.2 Q<RANGE[E] or (Q=RANGE[E] and USES[s]>1)

temp				*temp	
+		-		+	-
MOVE r,s		MOVN r,s		—	MOVNS s "negate L"
cons	lit	cons	lit	MOVE r,L(s)	
ADD r,C	ADDI r,L	SUB r,C	SUBI r,L		

ADDM r,L(t)

C.2 Code Matrix for (+,E1,E2,T)

1. MEM/MEM

```
MOVE   r,{*}E1
ADD    r,{*}E2
```

2. MEM/NUM

E2 not *	E2 *
MOVE r,{*}E1 "associate Num with T"	same as (1)

3. MEM/REG(s)

3.1 I_2 set

'reset temp mode'
'E1 ↔ E2'
'same as (13)'

3.2 E2 a temp

Q<RANGE[E2]		Q=RANGE[E2]	
MOVE r,s		+	-
+	-	ADD s,{*}E1	SUB s,{*}E1
ADD r,{*}E1	SUB r,{*}E1		

NB[T] ← NB[E2]

3.3 E2 not a temp

no temp associated with E2	temp associated with E2
ADD s,{*}E1	MOVE r,s ADD r,{*}E1

NB[T] ← '+'
"associate s with T"

4. MEM/REG(s)+NUM

'E1 ↔ E2'
'same as (13)'

5. NUM/MEM

'E1 ↔ E2'
'same as (2)'

6. NUM1/NUM2

6.1 I_f set: same as (2)

6.2 "associate NUM1+NUM2 with T"

7. NUM/REG(s)

7.1 I_f set: same as (3)

7.2 'E1 ↔ E2'
'same as (10)'

8. NUM1/REG(s)+NUM2

8.1 I_f set

'E1 ↔ E2'
'same as (13)'

8.2 Q<RANGE[E2] or (Q=RANGE[E2] and USES[s]>1)

temp		*temp	
MOVE r,s		+	-
+	-	——	MOVNS s "negate L"
INFO[T]←NUM1+NUM2	INFO[T]←NUM2-NUM1	MOVE r,L(s) "associate NUM1 with T"	
NB[T]←NB[E2]			

"associate r with T+NUM"

8.3 Q=RANGE[E2] and USES[s]≠1

same as (8.2) except replace r with s and delete the 'MOVE r,s' instruction

9. REG(s)/MEM

'E1 ↔ E2'
'same as (3)'

10. REG(s)/NUM

10.1 I_2 set

'E1 ↔ E2'
'same as (3)'

10.2 I_1 set

'reset mode of temp'
'same as (14)'

10.3 E1 a temp

Q<RANGE[E1]		Q=RANGE[E1]	
+	-	+	-
MOVE r,s	MOVN r,s	"associate NUM with T"	"associate -NUM with T"
"associate NUM with T+NUM" "associate r with T+NUM" NB[T]←'+'		"associate s with T+NUM" NB[T]←NB[E1]	

10.4 E1 not a temp

no associated temp	associated temp	
"assoc. s with T+NUM"	mode of reg="TEMP"	mode of REG="T+NUM"
	MOVE r,s "assoc. r with T+NUM"	"assoc. s with T+NUM"

"associate NUM with T"

11. REG(t)/REG(s)

11.1 I_1 set

'reset mode of $temp_1$'
'same as (15)'

11.2 I_2 set

'reset mode of $temp_2$'
'E1 ↔ E2'
'same as (15)'

11.3 E1 and E2 temps

a) Q<RANGE[E1] and Q<RANGE[E2]

MOVE r,t
'R ← r
E ← s
SW ← 1'

b) Q<RANGE[E1] and Q=RANGE[E2]

'R ← s
E ← t
SW ← 0'

c) Q=RANGE[E1] and Q<RANGE[E2]

'R ← t
E ← s
SW ← 1'

d) Q=RANGE[E1] and Q=RANGE[E2]

'R ← t
E ← s
SW ← 1'

E2 \ E1	+	-
+	ADD R,E NB[T]←'+'	SUB R,E SW=0: NB[T]←'-'; SW=1: NB[T]←'+'
-	SUB R,E SW=0: NB[T]←'+'; SW=1: NB[T]←'-'	ADD R,E NB[T]←'-'

"associate R with T"

11.4 E1 a temp and E2 not a temp

'E1 ↔ E2'
'same as (11.5)

11.5 E1 not a temp

11.5.1 temp associated with t: same as (3)

11.5.2 no temp associated with t

E2 not a temp	E2 a temp	
ADD t,s	+	-
	ADD t,s	SUB t,s

NB[T]←'+'

12. REG(t)/REG(s)+NUM

'E1 ↔ E2'
'same as (15)'

13. REG(s)+NUM/MEM

13.1 Q<RANGE[E1] or (Q=RANGE[E1] and USES[s]>1)

E2	temp		*temp	
	MOVE r,s		E1 +	-
E1 +		-	—	MOVNS s "negate L"
ADD r,{*}E1		SUB r,{*}E1	MOVE r,L(s) ADD r,{*}E1	
NB[T]↔NB[E1]			NB[T]←'+'	
"associate r with T+NUM" "associate NUM with T"			"assoc. r with T"	

13.2 Q=RANGE[E1] and USES[r]=1

same as (11.1) but replace r with s and remove the 'MOVE r,s' instruction

14. REG(s)+NUM1/NUM2

14.1 I_2 set: same as (13)

14.2 'E1 ↔ E2'
'same as (8)'

15. REG(t)+NUM/REG(s)

15.1 I_2 set

'reset mode of temp'
'same as (16)'

15.2 E2 a temp

15.2.1 Q<RANGE[E2] and (Q<RANGE[E1] or (Q=RANGE[E1] and USES[t]>1))

E1 \ E2	+	-
+	ADD r,s NB[T]←'+'	SUB r,s NB[T]←'-'
-	SUB r,s NB[T]←'+'	ADD r,s NB[T]←'-'

"associate r with T+NUM"

MOVE r,t

*temp

E1	+	-
	——	MOVNS t "negate L"

MOVE r,L(t)

E2	+	-
	ADD r,s	SUB r,s

"associate r with T"

15.2.2 Q<RANGE[E2] and (Q=RANGE[E1] and USES[t]=1)

same as (15.2.1) except replace r with t and delete the 'MOVE r,t' instruction

15.2.3 Q=RANGE[E2] and (Q<RANGE[E1] or (Q=RANGE[E1] and USES[t]>1))

Ei	temp		*temp	
E1 / E2	+	-	E2 +	-
t	ADD s,t NB[T]←'+' "NUM∿T"	SUB s,t NG[T]←'+' "-NUM∿T"	——	MOVNS t "negate L"
-	SUB s,t NB[T]←'-' "-NUM∿T"	ADD s,t NB[T]←'-' "NUM∿T"	E1 +	-
"associate s with T+NUM"			ADD s,L(t) NB[T]←'+' "associate NUM with T" "associate s with T"	SUB s,L(t) NB[T]←'-'

15.2.4 Q=RANGE[E2] and (Q=RANGE[E1] and USES[t]≠1)

E2	*temp	temp
	same as (15.2.3)	same as (15.2.2)

15.3 E2 not a temp

15.3.1 temp associated with s: same as (13)

15.3.2 no temp associated with s

E1	temp		*temp	
E1 +		-	E1 +	-
ADD s,t "NUM∿T"		SUB s,t "-NUM∿T"	——	MOVNS t "negate L"
"s∿T+NUM" NB[T]←'+'			ADD s,L(t) "s∿T" NB[T]←'+'	

16. REG(t)+NUM1/REG(s)+NUM2

Let $\beta_1 \equiv$ (Q<RANGE[E1] or Q=RANGE[E1] and USES[t]>1) and
(Q<RANGE[E2] or Q=RANGE[E2] and USES[s]>1)

$\beta_2 \equiv$ (Q<RANGE[E1] or Q=RANGE[E1] and USES[t]>1) and
(Q=RANGE[E2] and USES[s]≠1)

$\beta_3 \equiv$ (Q=RANGE[E1] and USES[t]=1) and
(Q<RANGE[E2] or Q=RANGE[E2] and USES[s]>1)

$\beta_4 \equiv$ (Q=RANGE[E1] and USES[t]≠1) and
(Q=RANGE[E2] and USES[s]=1)

16.1 E1 and *temp, E2 and * temp

E1 +	-
——	MOVNS t "negate E1"

E2 +	-
——	MOVNS s "negate E2"

16.1.1 β_1

```
MOVE  r,L1(t)
ADD   r,L2(s)

"associate r with T"
NB[T]←'+'
```

16.1.2 β_2

```
MOVE  s,L2(s)
ADD   s,L1(t)

"associate s with T"
NB[T]←'+'
```

16.1.3 β_3

```
MOVE  t,L1(t)
ADD   t,L2(s)

"associate t with T"
NB[T]←'+'
```

16.1.4 β_4: same as (16.1.3)

16.2 E1 not an *temp

16.2.1 β_1

E2	not an * temp		*temp
	MOVE r,s		E2 + | -
E1 \ E2	+	-	—— | MOVNS s "negate L"
+	ADD r,t INFO[T]←NUM1+NUM2 NB[T]←'+'	SUB r,t INFO[T]←NUM2-NUM1 NB[T]←'-'	E1 + | -
-	SUB r,t INFO[T]←NUM2-NUM1 NB[T]←'+'	ADD r,t INFO[T]←NUM1+NUM2 NB[T]←'-'	MOVE r,L(s) | MOVN r,L(s) ADD r,t "associate r with T+NUM1" "associate NUM1 with T"
	"associate r with T+NUM"		

16.2.2 β_2: same as (16.2.1) except replace r with s and remove the 'MOVE r,s' instruction

16.2.3 β_3

E2	not an *temp		*temp	
E1 \ E2	+	-	E2 +	-
			—	MOVNS s "negate L"
+	ADD t,s INFO[T]←NUM1+NUM2 NB[T]←'+'	SUB t,s INFO[T]←NUM1-NUM2 NB[T]←'+'	E1 +	-
			ADD t,L(s)	SUB t,L(s)
-	SUB t,s INFO[T]←NUM1-NUM2 NB[T]←'-'	ADD t,s INFO[T]←NUM1+NUM2 NB[T]←'-'	"associate t with T+NUM1" "associate NUM1 with T"	
	"associate t with T+NUM"			

16.2.4 β_4

E2 not an *temp	*temp
same as (16.2.2)	same as (16.2.3)

16.3 E1 an * temp, E2 not an *temp

'E1 ↔ E2'
'same as (16.2)'

References

All69 Allen, F.E. Program Optimizations. In **Annual Review in Automatic Programming**, Vol. 5, Pergamon, New York (1969), 239-307.

All70 Allen, F.E. Control Flow Analysis. In ACM Proceedings of a Symposium on Compiler Optimization, SIGPLAN Notices 5, 7 (JULY 1970), 1-19.

ASF66 USASI FORTRAN. American Standards Association Inc., New York, (March 1966).

Bli71 BLISS Reference Manual. Computer Science Report, Carnegie-Mellon University (Oct. 1971).

Cha67 Chartres, B.A. Algorithm 311 Prime Number Generator 2. **Comm. ACM 10**, 9 (Sept. 1967), 570.

Coc70 Cocke, J. and Schwartz, J.T. Programmming Languages and their Compilers. Preliminary notes (Second Edition). Courant Institute of Mathematical Sciences, New York University, (April 1970).

Dar70 Darden, Stephen C. and Heller, Steven B. Streamline your Software Development. **Computer Decisions 2** (Oct. 1970), 29-33.

For67 Forsythe, G.E. and Moler, C.B. **Computer Solution of Linear Algebraic Systems**. Prentice-Hall, Englewood Cliffs, N.J., (1967).

Gre69 Gegory, R.T. and Karney, D.L. **A Collection of Matrices for Testing Computational Algorithms**, Wiley-Interscience, New York, (1968).

Gri71 Gries, D. **Compiler Construction for Digital Computers**. John Wiley and Sons, Inc. (1971).

Har69 Harary, F. **Graph Theory**. Addison-Wesley Co. (1969).

Hay71 Haynam, G.E., Hansen, G.J. and Cook, R.P. A Tutorial on one-pass Compiler Design. XDS User's Group 16th International Meeting, Vol 2 (May 1971).

Ing71 Ingalls, D. The Execution Time Profile as a Programming Tool. In **Design and Optimization of Compilers**, edited by R. Rustin, Prentice-Hall (1971), 107-128.

Jas71 Jasik, S. Monitoring Execution on the CDC 6000's. In **Design and Optimization of Compilers**, edited by R. Rustin, Prentice-Hall (1971), 129-136.

Knu70 Knuth, D.E. An Empirical Study of FORTRAN Programs. IBM Report RC 3276 (1970).

Low69 Lowry, E. and Medlock, C.W. Object Code Optimization. **Comm. ACM 12**, 1 (Jan. 1969), 13-22.

McW72 McWilliams, T.M. Private correspondence (1972).

Mit70 Mitchell, J.G. Design and Construction of Flexible and Efficient Interactive Programming Systems. Ph. D. thesis, Carnegie-Mellon University, (June 1970), AD712721.

Mol72 Moler, C.B. Algorithm 423 Linear Equation Solver. **Comm. ACM 15**, 4 (April 1972), 274.

Mol73 Moler, C.B. and Stewart, G.W. An Algorithm for Generalized Matrix Eigenvalue Problems. **SIAM J. Numer. Anal. 19**, 2 (April 1973), 241-256.

PDP71a PDP10 Reference Manual. Digital Equipment Corp., Maynard, Massachusetts (1971).

PDP71b PDP10 Timesharing Handbook. Digital Equipment Corp., Maynard, Massachusetts (1971).